Electronic Notes in Theoretical Computer Science 50 No. 2 (2001) – Proc. BOTH 2001
URL: http://www.elsevier.nl/locate/entcs/volume50.html 19 pages

A CuCh Interpretation of an Object–Oriented Language [1]

Viviana Bono Ivano Salvo

Dipartimento di Informatica, Università degli Studi di Torino
corso Svizzera 185, 10149 Torino, Italy,
e-mail: {bono,salvo}@di.unito.it

Abstract

Böhm's CuCh machine extends the pure lambda–calculus with algebraic data types and provides the possibility of defining functions over disjoint sums of algebras. We exploit such natural form of *overloading* to define a functional interpretation of a simple, but significant fragment of a typical object–oriented class–based language.

Dedicated to Professor Corrado Böhm
Recipient of the EATCS Distinguished Service Award
Maestro and Inspirer

1 Introduction

The theoretical research on modelization of object–oriented languages has been developed following two main directions: the definition of object–oriented calculi (for example [26,20,1,19]), where the notion of either "object" or "class" (or both) is taken as primitive, and the encoding of object–oriented components inside typed (starting from [21]) and untyped (starting from [24]) lambda–calculi.

The aim of both approaches is to grasp what essentially characterizes the object–oriented paradigm: in the former by singling out a restricted number of well–chosen basic constructs, possibly expressive enough to model the essence of object orientation, in the latter by interpreting object features via well–understood concepts.

While most widely used object–oriented programming languages are class-based (such as C^{++} [25] and Java [3]), the majority of the theoretical studies is about object-based formal systems, without classes and often without an

[1] Partially supported by MURST Cofin '99 TOSCA.

inheritance mechanisms. Classes have been studied either as a primitive construct ([20,19,16,30]), or as a combination among extensible objects and traditional data abstraction ([27,15]), but they are not as well-understood from a mathematical point of view as objects are instead. In fact, lots of different translations of object-based calculi into different typed lambda–calculi have been proposed [29,2,14] (a detailed comparison of various encodings can be found in [18]), which highlight the mathematical meaning of objects.

We think that a promising step forward in understanding the class concept may lay in further exploring connections between object–oriented and functional paradigms, even though object-orientation was born from the imperative paradigm. In particular, the functional setting we consider, the *Cuch machine* [8,13], offers algebraic data types and the possibility of defining functions over disjoint sums of algebras (and hence a natural form of overloading), so it seems a natural setting to be extended in order to model concepts as classes, inheritance, dynamic method lookup and information hiding. Starting from Böhm's intuition about relationships between the *data–driven* programming style of the CuCh machine and the object–oriented programming style, we will study how to encode object–oriented constructs into the CuCh machine, taking advantage of the CuCh features as described in Section 2. To formalize our intuitions, we introduce a toy class-based language CuCh^{++} which embodies typical features of (untyped) class-based languages and we show how to encode it into the CuCh machine. The basic ideas of our encoding is to interpret classes as algebras, object constructors as algebraic constructors, objects as terms of such algebras, and methods as recursively defined functions, and to take advantage of the possibility of defining overloaded functions in order to manage overriding and to mimic dynamic method lookup inside lambda–calculus. As an interesting outcome of our preliminary investigations we have that some of the most common concepts of object–oriented languages, such as inheritance, dynamic method lookup and a primitive form of encapsulation (public methods and private fields) are naturally interpreted in the language CuCh.

In this paper we present the basic CuCh^{++} ideas and the formal definition of our CuCh^{++}–to–Cuch encoding, but we defer to future work its metatheoretical analysis. In Section 5 we will briefly outline how to address the problem of proving a form of correctness for the CuCh^{++}–to–CuCh encoding.

The rest of the paper is organized as follows: in Section 2 we hint at the seminal Böhm's CuCh machine, highlighting those aspects which inspired this work. In Section 3, we introduce the syntax and the (informal and operational) semantics of CuCh^{++}. In Section 4, we explain which basic ideas are behind the CuCh^{++}–to–CuCh encoding, by illustrating a variation of the canonical "point/colored point" example, and then we give its formal definition. Section 5 concludes the paper: we discuss there further extensions of this paper and related work.

2 The CuCh Machine

The CuCh machine implements the functional programming language CuCh which was one of the earlier "concrete proposals" for a pure functional programming language [6,12,10,11].

The well-known paper by Backus [4] proposed a purely functional language based essentially on combinators, as showed in [7], as an alternative to imperative programming languages which are related rather to the von Neumann's architecture than to problem specifications. This stimulated a new interest in making functional programming languages rich enough to be employed into the development of concrete applications. Particular interest was devoted to embed algebraic data types and relative mappings of algebras within lambda–terms and this research is still on [5]. A methodology to represent any type of term algebras and functions "iteratively" defined on that algebras, using second order typed normal forms, was introduced by Böhm and Berarducci in [9].

In [13], CuCh was extended, allowing to embed arbitrary algebraic data types and mappings and to define recursive functions over such algebras, by systems of recursive equations. The main idea of CuCh is to solve such systems of equations *inside* lambda–calculus, in such a way that functions are interpreted as *normal-form* lambda-terms.

More precisely, let Σ be a signature, partitioned into two sets, $\Sigma_0 = \{c_1, \ldots, c_l\}$ (data constructors) and $\Sigma_1 = \{f_1, \ldots, f_k\}$ (function symbols). Given a set of equations:

$$f_i(c_j\ x_1\ \ldots\ x_n)y_1 \ldots y_m = b_{i,j}$$

where $1 \leq i \leq k, 1 \leq j \leq l$ and $b_{i,j}$ an *extended lambda-term* with symbols from the signature Σ, it is possible to find normal solutions to this system, i.e., normal-form lambda-terms that, once substituted for f_i and c_j, satisfy each equation.

In the CuCh machine there are two modes, `@lambda` and `@env`: the former is essentially an assignment language allowing to reduce functional expressions which includes *Cu*rry's combinators and *Ch*urch's lambda terms, whereas the latter allows definitions of data belonging to arbitrary (free) term algebras, and recursive definitions of functions whose first argument belongs to such an algebra. Algebras are defined specifying the constructors and their arity. Functions are defined by systems of recursive equations. An interesting aspect is that functions can be defined over the *union* of term algebras, so that functions are usually *overloaded*. This feature will play an essential role in the CuCh^{++}–to–Cuch encoding.

Example 2.1 Here we show how to define the algebras of natural numbers and (polymorphic) lists[2] and how to define an overloaded function `sum` which

[2] In the latest version of the CuCh machine, natural numbers and lists are built-in datatypes. The abbreviations `n` for `succ`^{n} `zero` and `[h1, h2, ...hn]` for `cons h1 (cons h2 ...(cons hn nil) ...)` were introduced for pragmatic reasons.

either adds two integers or appends two lists according to the arguments the function is applied to.

```
@env;
Nat := {succ:1, zero:0};
List := {cons:2, nil:0};
sum zero m := m;
sum (succ n) m := succ (sum n m);
sum nil l := l;
sum (cons a tl) l = cons a (sum tl l);
```

The CuCh machine solves this system of recursive equations inside the lambda–calculus, finding a map which associates to each symbol `sum, cons, nil, succ, zero` a lambda-term such that the equations are satisfied.

In the `@lambda` mode we can evaluate expressions like the following ones:

```
@lambda;
l := cons 12 (cons 5 (cons 3 nil)));
m := cons 4 (cons 6 nil);
o := sum l m;
cons 12 (cons 5 (cons 3 (cons 4 (cons 6 nil))))
s := sum (succ zero) (succ (succ zero))
succ (succ (succ zero))
```

It is worthwhile to single out two main points:

- lambda-terms which are solutions of this system of recursive equations are normal forms and this distinguishes the CuCh treatment for recursive function definitions from the usual treatment via fixed point combinators: here functions (programs) are normal forms which, when applied to some arguments, incidentally may diverge;
- the engine of the recursion is data, so we may talk of data–driven programming, and this can be seen as an analogue of object–oriented programming, where objects, regarded as data, drive the computation when they reply to method invocations by executing the appropriate code.

3 The Language CuCh^{++}

In this section we define the syntax of the calculus CuCh^{++}. As mentioned in the Introduction, CuCh^{++} can be seen as a distilled class-based language, designed with the main purpose to formalize our intuitions about relationships between the CuCh programming style and the object–oriented programming style, while keeping in mind the prominent features of class-based languages. Furthermore, it also provides a core language for an extension of the CuCh machine. To this aim, a preliminary version of CuCh^{++} was implemented [32].

CuCh^{++} is an extended lambda–calculus, where programs are expressions evaluated in an *environment* declared as a **let** statement. Such environment

contains the class definitions. We choose not to include classes as first-order entities because this makes easier to deal with class *names*, which, in turn, make easier defining the encoding of Section 4 where classes are seen as algebras (moreover, this is not a totally uncommon practice [3], see for example [30]). So, an object is generated by applying a *new* operation to a class name. A class definition specifies the superclass' class, together with the behavior of its objects by listing the fields and the method names (and respective bodies). A class silently inherits all methods and fields from its superclass. We assume, for the sake of uniformity, that each class has always a superclass, and for this purpose we introduce a top class, the empty class *Empty*, in which neither fields nor methods are defined. The visibility rules for this first release of the CuCh^{++} are inspired from a sort of "purist" object–oriented point of view, in which fields are *private* (i.e, accessible only via methods defined inside the class) and methods are *public* (i.e., accessible everywhere).

We consider an extended lambda–calculus equipped with a set of constants $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \Sigma_2$, from where we choose respectively class, field and method names. In the CuCh^{++} syntax we distinguish between *class declarations*, indicated by the symbol D, and *expressions*, indicated by the symbol e.

$$
\begin{array}{l}
D ::= \textbf{class}\ C\ \textbf{is}\ C' \\
\qquad \{\ v_1,\ \ldots\ , v_k, \\
\qquad\ \ \textbf{meth}\ m_1\ = e_1, \\
\qquad\qquad \vdots \\
\qquad\ \ \textbf{meth}\ m_n\ = e_n\ \}
\end{array}
$$

where $C, C' \in \Sigma_0$, $v_1, \ldots, v_k \in \Sigma_1$, and $m_1, \ldots, m_n \in \Sigma_2$. The keyword **is** introduces the superclass C' of the class C.

$$P ::= \textbf{let}\ D_1 \ldots D_n\ \textbf{in}\ e \quad n \geq 0$$

where e is an expression defined as follows.

e	::=	x	variable
		$\lambda x.e$	abstraction
		$(e_1 e_2)$	application
		$e \Leftarrow m$	method invocation
		new C	object creation
		$e_1 \# v = e_2$	field update
		$e.v$	field selection
		self	
		super $\Leftarrow m$	

where x is a variable, $C \in \Sigma_0, m \in \Sigma_2$. We assume usual lambda notations and conventions about function application and variable name renaming. The intended intuitive meaning of the object–oriented constructs introduced above

[3] Having named classes is useful in a typed setting when the *subtyping by name* is used, but it make sense in our untyped setting, too, since we have here the untyped counterpart of such subtyping, i.e., *substitutivity* [28].

is the following:

$e \Leftarrow m$ denotes the invocation of the method m on o, if e reduces to an object o;

new C creates an object that is an instance of the class C.

$e_1 \# v = e_2$ denotes the object o where the content of the field v in o is replaced with e_2, if e_1 reduces to an object o;

$e.v$ denotes the selection of the value associated to the field v in o, if e reduces to an object o;

self is a reference to the host object inside a method body definition. As usual, its meaning is dynamic, i.e., the host object may belong to a subclass of the class where the method is defined;

super allows the previously defined code of an overridden method to be referred. **super** $\Leftarrow m$ has a static meaning (as it has in the most common object–oriented languages, from Smalltalk [28] to Java [3]), i.e., **super** $\Leftarrow m$ fetches the body of m starting the lookup from the superclass of the class whose the method containing the **super** invocation belongs to. We observe that **super** does not influence the binding for **self**;

The basic object operations (field update and field selection) can be performed only if contained in a method body. Moreover, there are expressions which are not included in the source calculus, but can be generated during the evaluation process. These are the objects, created via **new** operations on class names. An object has a record-like shape: $\langle v_1 = e_1, \ldots, v_k = e_k, m_1 = b_1, \ldots, m_n = b_n, parent = e \rangle$, where v_i's are its fields and m_j's are its methods. When k is 0 we have an object without fields and when n is 0 the object has no methods attached. An object generated by the top class *Empty* has then the shape: $\langle\rangle$. Observe that an object is a nested record, since the expression associated to the special field *parent* is an object itself. Informally speaking, *parent* contains information about the superclass of the object's class. The precise meaning and use of *parent* will be clear within the operational semantics rules. Notice also that a field can contain an object as its value, and in particular also an object generated by the same class whose the object belongs to. Using this fact, it is possible to define inductive data types, like natural numbers or lists, via class definitions.

Given the two above mentioned restrictions, we can now define what a legal CuCh^{++} program is.

Definition 3.1 [Programs] A declaration **let** $C_1 \ldots C_n$ **in** e is a CuCh^{++} *program* if e is generated by the above pseudo-grammar and:

(i) keywords **self**, **super** and operators . and # appear only inside method body definitions.

(ii) objects do not appear in e.

3.1 Operational Semantics

$$(\textbf{let}) \quad \frac{e \rightsquigarrow_{\{D_1,\ ...,\ D_n\}} e'}{\textbf{let}\ D_1,\ \ \ldots,\ D_n\ \textbf{in}\ e \rightsquigarrow_{\varnothing} e'}$$

$$(\beta) \quad \frac{e_1 \rightsquigarrow_\Gamma (\lambda x.e_1')}{(e_1 e_2) \rightsquigarrow_\Gamma e_1'[x := e_2]}$$

$$(\Leftarrow) \quad \frac{e \rightsquigarrow_\Gamma o \qquad o = \langle v_i = e_i, m_j = b_j, parent = e' \rangle \qquad m = m_{j_0}}{e \Leftarrow m \rightsquigarrow_\Gamma (o.m\ o)}$$

$$(upd_1) \quad \frac{e_1 \rightsquigarrow_\Gamma o \qquad o = \langle v_i = e_i, m_j = b_j, parent = e' \rangle \qquad v = v_{i_0}}{e_1 \# v = e_2 \rightsquigarrow_\Gamma \langle v_i = e_i (i \neq i_0), v_{i_0} = e_2, m_j = b_j, parent = e' \rangle}$$

$$(upd_2) \quad \frac{e_1 \rightsquigarrow_\Gamma o \qquad o = \langle v_i = e_i, m_j = b_j, parent = e' \rangle \qquad v \neq v_i \forall i}{e_1 \# v = e_2 \rightsquigarrow_\Gamma \langle v_i = e_i, m_j = b_j, parent = (e' \# v = e_2) \rangle}$$

$$(ext_1) \quad \frac{e \rightsquigarrow_\Gamma o \qquad o = \langle n_i = e_i, parent = e' \rangle \qquad n = n_{i_0}}{e.n \rightsquigarrow_\Gamma e_{i_0}}$$

$$(ext_2) \quad \frac{e \rightsquigarrow_\Gamma o \qquad o = \langle n_i = e_i, parent = e' \rangle \qquad \forall i \quad n \neq n_i}{e.n \rightsquigarrow_\Gamma e'.n}$$

$$(\textbf{new}_1) \quad \frac{\begin{array}{c}\textbf{class}\ C\ \textbf{is}\ C'\{v_1, \ldots, v_k, \textbf{meth}\ m_1 = b_1 \ldots \textbf{meth}\ m_n = b_n\} \in \Gamma \\ \textbf{new}\ C' \rightsquigarrow_\Gamma \lambda y_1 \ldots y_l.e' \equiv e'' \\ x_1, \ldots, x_k, z_1, \ldots, z_l \text{ fresh variables}\end{array}}{\textbf{new}\ C \rightsquigarrow_\Gamma \lambda x_1 \ldots x_k z_1 \ldots z_l . \langle v_i = x_i, m_j = b_j, parent = (e''\ z_1 \ldots z_l) \rangle}$$

$$(\textbf{new}_2) \quad \frac{}{\textbf{new}\ Empty \rightsquigarrow_\Gamma \langle\rangle}$$

Fig. 1. Operational Semantics of CuCh^{++}

$$\frac{\begin{array}{l}\textbf{class}\ C\ \textbf{is}\ C'\{v_1, \ldots, v_k, \\ u_1 = e_1, \ldots, u_r = e_r, \\ \textbf{meth}\ m_1 = b_1 \ldots \textbf{meth}\ m_n = b_n\}\end{array} \in \Gamma \qquad \textbf{new}\ C' \rightsquigarrow_\Gamma \lambda y_1 \ldots y_l.e'}{\begin{array}{c}\textbf{new}\ C \rightsquigarrow_\Gamma \begin{array}{l}\lambda x_1 \ldots x_k y_1 \ldots y_l. \\ \langle v_i = x_i, u_h = e_h, m_j = b_j, \\ parent = (e'\ y_1 \ldots y_l) \rangle\end{array}\end{array}}$$

Fig. 2. An alternative rule for **new**

Figure 1 shows the formal definition of the operational semantics for the language CuCh^{++}. The operational reduction relation $\rightsquigarrow_\Gamma$ is parameterized

with respect to an environment Γ, built up from the class declarations (rule (**let**)). The environment Γ is essentially needed to reduce **new** expressions.

We assume that the keyword **self** is only syntactic sugar standing for a bound variable *self*, i.e., all method bodies b are abstractions of at least the *self* parameter and so the method bodies we consider in the operational semantics have the shape $b' = \lambda self.b[self/\mathbf{self}]$.

As remarked in the previous section, even though **super** is treated in the syntax as if it were an object, it is not really an object in the usual sense [31]: giving the name **super** as the receiver of a message indicates only where to start the method lookup. So, **super** denotes a message environment that can be statically determined. With this idea in mind, we assume to pre-process a CuCh^{++} program before executing it, in order to substitute each expression of the form $\mathbf{super} \Leftarrow m$ with the expression $b_m\mathbf{self}$, where b_m is the appropriate method body found up in the class hierarchy. We apply it to **self** in order to preserve the right host object when the program will be executed.

We are ready now to describe the reduction rules.

We have the usual (call-by-name) (β) rule of lambda–calculus. The method invocation rule ($\Leftarrow$) looks up for the appropriate method body by triggering the ($ext\_$) rules. Rule (ext_1) takes care of the case when a method was newly introduced by the class of the receiver. Rule (ext_2), instead, looks up for the method in the hierarchy, by re-triggering the search on the *parent* content, which contains fields and methods of the superclass, and a further reference up to the hierarchy. Then, the method body is applied to the host object (as in the classical *self-application* semantics [1]). Rules (ext_1) and (ext_2) are also used for the retrieval of a field value. Rules (upd_1), (upd_2) deal with the replacement of a field value and they basically work as the ($ext\_$) rules in order to retrieve the right field to be updated. In rules ($ext\_$) and ($upd\_$), we use the metavariable n to range over both field and method names. Rule ($\mathbf{new}_1$) shows how objects are instantiated from classes. A **new** expression evaluates to a function whose parameters, $x_1, \ldots, x_k, z_1, \ldots, z_l$ are the initial values for the fields (both new, the x_i's, and inherited, the z_j's), so that actual values must be supplied. Thanks to the special field *parent*, the initialization phase for the superclass' fields is performed without having to explicitly mention their names. Delegating the superclass to deal with its own fields is good design practice (see [27]), and even though in our case a class must be aware of the number of fields of its immediate superclass, this is the only information leakage. Rule ($\mathbf{new}_2$) deals with the special case of instantiating the top class *Empty*.

We observe that we could easily modify the language and the operational semantics by providing the possibility of associating an initial value to a field in the class definition. In such a case the rule (**new**) should be rewritten as in Figure 2.

4 The Translation

In this section we define our translation of CuCh^{++} programs into CuCh programs. First we informally describe some basic ideas, then we present an example, and finally we give the formal definition of the translation.

In our encoding, we avoid the use of explicit fixed points (as instead, for example, in the inheritance denotational model of Cook [24]), on the same wavelength as in the CuCh philosophy. Each class definition is translated basically as an algebra definition with one data constructor: the arity of such constructor depends on the number of fields. Objects are elements of a (freely generated) algebra, which record the data structure (the fields) of the object. Method bodies are translated into equations that define functions recurring on their first parameter, which belongs to the algebra representing the class. Method invocation is translated simply as function application.

The mechanism of the Cuch machine for defining functions over the union of algebras allows to associate different "pieces of code" to the same function, depending on the data constructor of the first argument (overloading). Therefore, translation of overriding methods is straightforward thanks to this feature: it suffices to translate the overriding method body. An inherited method is translated by extending the definition of the function corresponding to the superclass method with an equation whose argument is a subalgebra element, and such equation works as a mere reference to the superclass method code. Once again it is overloading that makes everything work correctly.

A translation of a class definition produces also the definition of a CuCh function called `new` interpreting the **new** operation of CuCh^{++}. In order to have it overloaded, we introduce a special algebra `Classes` with a 0-ary constructor for each class.

Point, Colored Point and Bidimensional Point

In this subsection, we give the definitions of three classes in CuCh^{++}, Point, C_Point and B_C_Point, show how these definitions are translated into Cuch language, and briefly discuss how our translation manage with inheritance, method specialization, method override, and dynamic method lookup.

Notations and Syntactic Sugar.

We use a name convention which links class names in the source language with constructors and names of algebras in the target language (see below the encoding of class definitions). To distinguish CuCh^{++} programs from their CuCh translations, we use a typewriter font for the latter. For the sake of simplicity, we assume that both in CuCh^{++} and CuCh usual arithmetic and logical operators (like $+$, $<$, $=$, and *abs* for absolute value function) are built-in and prefixed. Moreover, we freely use in both CuCh^{++} and CuCh examples some syntactic sugar such as **if** b **then** e_1 **else** e_2 statements (`if b` $\mathtt{e}_1$ $\mathtt{e}_2$ in CuCh).

The definition of the Point class defines the behavior of a one-dimensional point. The field x keeps information about the position of the point, the method *get_x* reads such position, the method *dist* calculates the distance of the point from the origin and the method *move* changes this position, given a displacement argument.

class Point is Empty{x;
meth $get\_x = \mathbf{self}.x$
meth $move = \lambda d.\mathbf{self}\#x = (+\ \mathbf{self}.x\ d)$
meth $dist = abs\ \mathbf{self}.x$
}

C_Point (subclass of Point class) defines the behavior of a colored point. A field c keeps information about the color and the *get_c* method allows such information to be accessed. The method *set_color* always sets the color of the point to "BLACK", if the point is outside a fixed interval.

class C_Point **is** Point {c;
meth $get\_color = \mathbf{self}.c$
meth $set\_color = \lambda col.\mathbf{if}(<(\mathbf{self} \Leftarrow dist)\ 3)$
then self$\#c = col$
else self$\#c =$ BLACK
}

Let us now define the class of bidimensional point, B_C_Point, where we override methods *dist* and *move*:

class B_C_Point **is** C_Point {y;
meth $dist = max\ (abs\ \mathbf{self}.y)\ (\mathbf{super} \Leftarrow dist)$
meth $move = \lambda d_x d_y.(\mathbf{super} \Leftarrow move\ d_x)\#y = (+\ \mathbf{self}.y\ d_y))$
}

The translation of above definitions gives rise to the following CuCh definitions.

Classes.

First thing, we need to define a special algebra `Classes`, the purpose of which will be clearer when we discuss the translation of the **new** operation.

```
Classes:={POINT:0, C_POINT:0, B_C_POINT:0, EMPTY:0}
Point:={point:1}
C_Point:={c_point:2}
B_C_Point:={b_c_point:2}
Empty:={empty:0}
```

The algebra `Classes` has a 0-ary constructor for each class definition. An algebra with a set of 0-ary constructors allows the definition of functions over a finite domain and hence different equations (representing different code) to be associated to the `new` function, one for each class of the CuCh^{++} program, i.e., `new` is overloaded as wished before.

As the second step, for each class definition we generate an algebra definition.

The data constructor `c_point` (resp. `b_c_point`) has arity 2, because each colored point (resp. bidimensional colored point) keeps information about the color (resp. the second coordinate) and the information about the point (resp. the colored point). Each canonical element in the algebra `C_Point` has the shape `c_point c (point x)` and each canonical element in the algebra `B_C_Point` has the shape `b_c_point y (c_point c (point x))`.

Also the arity of the data constructor `point` is 2, since by uniformity we required a parent class for each class (see Section 3). Hence a translation of a point object is a canonical element of the algebra `Point` and it has the shape `point x empty`.

This encoding provides a direct way to access the resources of, for example, the "point" part of a colored point (as we want the superclass' methods taking care of its own fields).

Field selection and field update.

It turns out that to properly translate field selection and field update (# and . operators) we need to define a set of auxiliary functions[4]. For each field v (`v`) we define two Cuch functions, $\mathtt{s_v}$ and $\mathtt{u_v}$, and we translate each subexpression of the form $e_1\#v = e_2$ to $(\mathtt{u_v}\ \mathtt{e_1}\ \mathtt{e_2})$, where $\mathtt{e_1}$ (resp. $\mathtt{e_2}$) is the translation of e_1 (resp. e_2). Similarly, we translate subexpression of the form $e.v$ with $(\mathtt{s_v}\ \mathtt{e})$ where `e` is the translation of e. When we define a subclass M of a class N, we uniformly specialize the auxiliary functions on the elements of the $\mathtt{C_M}$ subalgebra simply by adding the corresponding equations, taking advantage of CuCh overloading.

```
u_x (point x parent) z = point z
s_x (point x parent) = x

u_c (c_point c parent) z = c_point z parent
s_c (c_point c parent) = c
u_x (c_point c parent) z = (c_point c (u_x parent z))
s_x (c_point c parent) = s_x parent

u_y (b_c_point y parent) z = b_c_point z parent
s_y (b_c_point y parent) = y
u_x (b_c_point y parent) z = (b_c_point y (u_x parent z))
s_x (b_c_point y parent) = s_x parent
u_c (b_c_point y parent) z = (b_c_point y (u_c parent z))
s_c (b_c_point y parent) = s_c parent
```

[4] The motivation is rather technical and it depends on the behavior of # and . when they are applied to expressions other than **self**.

Methods (newly defined and overriding).

We translate each newly defined and overriding method body into a function that takes as its first parameter an element of the algebra representing the class the method belongs to, this parameter leading the dynamic lookup process (supported by the overloading), and as its second parameter `self`, i.e., the host object. Having introduced auxiliary functions to deal with selecting and updating fields, the function which translates a method body does not need to directly access information inside elements of algebras. As a consequence, the only relevant information about the above mentioned first parameter is its algebra (for instance, `POINT` for the Point objects), but not its inner structure (for instance, `x`), which is taken care by the appropriate auxiliary functions `D`.

```
get_x POINT = lmb:  self.  s_x self
move POINT = lmb:  self d.  u_x self (+ (s_x self) d)
dist POINT = lmb:  self.  (abs s_x self)
get_color C_POINT = lmb:  self.  s_c self
set_color C_POINT = lmb:  self c.  u_c self c
move B_C_POINT = lmb:  self dx dy.
      u_y (move C_POINT self dx) (+ (s_y self) dy)
dist B_C_POINT = lmb:  self.
      max (abs (s_y self)) (dist C_POINT self)

D (point x parent) = POINT
D (c_point c parent) = C_POINT
D (b_c_point y parent) = B_C_POINT
```

Methods (inherited).

When we define the subclass C_Point, we expect that the *get_x* and *move* methods, defined in the class Point, to work properly on C_Point objects. To obtain this in the CuCh translation, we have to introduce equations which define appropriate functions over the `C_Point` algebra, i.e., equations that extend the `get_x` and `move` definitions on the elements of the subalgebra `C_Point`.

```
move C_POINT = move POINT
get_x C_POINT = get_x POINT
dist C_POINT = dist POINT

get_x B_C_POINT = get_x C_POINT
get_color B_C_POINT = get_color C_POINT
set_color B_C_POINT = set_color C_POINT
```

Observe that we do not introduce equations in order to make `move` and `dist` functions work on `B_C_Point` algebra elements. Methods *move* and *dist* are overridden in the B_C_Point Class definition, and hence defined as in the previous paragraph.

Object Generator Function.

In order to have an overloaded `new` operation (which implements the **new** operation of CuCh^{++}), we exploit the special algebra `Classes` introduced before. We have as many equations that define `new` in CuCh as many class definitions in the CuCh^{++} program. In our example, then, we obtain:

```
new EMPTY = empty
new POINT = lmb:x.point x (new EMPTY)
new C_POINT = lmb:c x.c_point c (new POINT x)
new B_C_POINT = lmb:y c x.c_point y (new C_POINT c x)
```

Expressions.

We now show an expression that can be evaluated in the environment built over the above class definitions:

$$((\textbf{new}\ \text{B\_C\_Point}\ 5\ \text{GREEN}\ 2) \Leftarrow \textit{set\_color}\ \text{RED}) \Leftarrow \textit{get\_color}$$

The evaluation of such expression gives as result BLACK because of dynamic method lookup: the invocation of the *dist* method in the body of *set_color* executes the body of the *dist* method defined in the B_C_Point class and hence it gives 5 as result. During the execution of *set_color*, **self** is in fact bound to a B_C_Point.

The CuCh translation of the above expression is:

```
get_color (D (set_color (new B_C_POINT 5 GREEN 2) RED) (set_color
   (D (new B_C_POINT 5 GREEN 2)) (new B_C_POINT 5 GREEN 2) RED)
```

and it reduces to `BLACK`, as it can be checked reducing this expression considering CuCh equations as a rewriting system.

Formal Definition of the Translation

Expressions

$\tau(x) = \mathtt{x}$
$\tau(\lambda x.e) = \mathtt{lmb:}\quad \mathtt{x}.\tau(e)$
$\tau(e_1\ e_2) = (\tau(e_1)\tau(e_2))$
$\tau(e \Leftarrow m) = \mathtt{m\ D}(\tau(e))\tau(e)$
$\tau(\textbf{new}\ M) = \mathtt{new\ M}$

$\tau(e_1 \# v\ e_2) = \mathtt{u_V}\tau(e_1)\tau(e_2)$
$\tau(e.v) = \mathtt{s_V}\tau(e)$
$\tau(\textbf{self}) = \mathtt{self}$
$\tau(\textbf{super} \Leftarrow m) = \mathtt{m\ C_N\ self}$

Translation of a Class Definition

The translation of a CuCh^{++} class definition of the shape:

class M **is** N
$\{\ v_1, \ \ldots\ , v_k,$
meth $m_1\ = e_1,$
$\vdots$
meth $m_l\ = e_l\ \}$

corresponds to a sequence of CuCh definition, as follows:

algebra definitions A definition of a free algebra with a data constructor of arity $k+1$, a 0-ary constructor $\mathtt{C_M}$ to be added in the algebra `Classes`:

$$\mathtt{M:=\{c_M:k+1\}} \qquad\qquad \mathtt{Classes:=\{\ldots,\ C_M:0\}}$$

and the auxiliary `D` function:

$$\mathtt{D\ (c_M\ v_1\ \ldots v_k)\ =\ C_M}$$

auxiliary functions For each field $\mathtt{v_i}$ the definition of the auxiliary functions $\mathtt{u_{v_i}}$ and $\mathtt{s_{v_i}}$ is as follows:

$$\begin{aligned}
&\mathtt{u_{v_1}\ (c_M\ v_1\ v_2\ \ldots v_k\ parent)\ z\ =\ c_M\ z\ v_2\ \ldots v_k\ parent}\\
&\qquad\vdots\\
&\mathtt{u_{v_k}\ (c_M\ v_1\ v_2\ \ldots v_k\ parent)\ z\ =\ c_M\ v_1\ v_2\ \ldots z\ parent}\\
&\mathtt{s_{v_1}\ (c_M\ v_1\ \ldots v_k\ parent)\ =\ v_1}\\
&\quad\vdots\\
&\mathtt{s_{v_k}\ (c_M\ v_1\ \ldots v_k\ parent)\ =\ v_k}
\end{aligned}$$

and the specialization of the auxiliary functions (defined in the translation of the parent class N) is as follows:

$$\begin{aligned}
&\mathtt{u_{y_1}\ (c_M\ v_1\ v_2\ \ldots v_k\ parent)\ z\ =}\\
&\qquad\qquad\mathtt{c_M\ v_1\ v_2\ \ldots v_k\ (u_{y_1}\ parent\ z)}\\
&\qquad\vdots\\
&\mathtt{u_{y_l}\ (c_M\ v_1\ v_2\ \ldots v_k\ parent)\ z\ =}\\
&\qquad\qquad\mathtt{c_M\ v_1\ v_2\ \ldots v_k\ (u_{y_l}\ parent\ z)}\\
&\\
&\mathtt{s_{y_1}\ (c_M\ v_1\ \ldots v_k\ parent)\ =\ s_{y_1}\ parent}\\
&\quad\vdots\\
&\mathtt{s_{y_l}\ (c_M\ v_1\ \ldots v_k\ parent)\ =\ s_{y_l}\ parent}
\end{aligned}$$

methods (newly defined and overriding) For each method m_i new or overriding we have:

$$\mathtt{m_i\ C_M\ =\ lmb:\ \ self.}\ \ \tau(e_i)$$

methods (inherited) For all functions $\mathtt{n_i}$ generated from the encoding of methods of N which are not overridden in M we add the equation:

$$\mathtt{n_i\ C_M\ =\ n_i\ C_N}$$

new Let's assume that the objects of N has j fields. The definition of the `new` function is then as follows:

$$\texttt{new } C_M = \texttt{lmb: } v_1 \ldots v_k\ y_1 \ldots y_j.\ c_M\ v_1 \ldots v_k\ (\texttt{new } C_N\ y_1 \ldots y_j)$$

5 Conclusions

We presented an (untyped) object–oriented calculus CuCh^{++} inspired by the Böhm's Cuch philosophy, and provided a CuCh^{++}–to–Cuch encoding that takes care of some of the main concepts of object–oriented programming languages, such as classes, objects, inheritance, dynamic method lookup and a primitive form of encapsulation (public methods and private fields). The encoding exploits one of the most prominent feature of CuCh, i.e., the data–driven computation style (supported by overloaded functions), and provides a semantics for CuCh^{++}. The CuCh^{++} calculus and the related encoding can then be seen from two orthogonal points of view: as a well-founded class-based calculus, simple yet representative, and as a novel functional interpretation of object–oriented features.

In an untyped scenario, our object encoding may provide an alternative formalization of classes and objects which avoids the use of explicit fixed points, as instead in [24,16]. It can be objected that a calculus based on overloading seems more an implementation rather than a formalization. This could be true, but there is evidence that some phenomena like method override are related more to a syntactic universe of "names", rather than to a semantic universe of functions. For example, in our encoding method specialization in the inheritance process has a clear meaning in terms of algebra morphisms, but overriding does not seem to fit in clear algebraic concepts. So it looks worthwhile further pursuing this line of research.

Related Work

A theoretical analysis of overloading has been carried out by Castagna, Ghelli and Longo in [23]: aimed by the purpose of providing a general framework as a foundation of object–oriented languages, they introduced an extension of simply typed lambda–calculus, $\lambda\&$, where overloaded terms and types are considered. In [22], an encoding of a toy object–oriented language into $\lambda\&$ has been proposed. In the $\lambda\&$-calculus, an overloaded term is essentially a collection of functional terms, glued together. When an overloaded term $M = M_1\& \ldots \& M_n$ is applied to an argument N, one *branch* M_i of the overloaded term M is chosen depending on the type of N, and then applied to N. Therefore, the reduction relation of this calculus (and hence its operational semantics) depends on types. In our approach, properties peculiar to the CuCh machine allow us to encode overloading in pure lambda–calculus, keeping the natural treatment of application and abstraction. Moreover, they allow us to avoid type-dependent operational semantics and to obtain a quite

natural interpretation of object–oriented features into an extension of pure lambda–calculus equipped with algebraic data types and, via the Interpretation Theorem, in the pure lambda calculus as well.

Another related piece of work is in [17], where the formal foundations of a design method integrating algebraic specification techniques and object–oriented programming is presented. This work is really interesting since it exploits algebraic data types definitions in a software-engineering setting. We still do not know whether if there is a possible use of the CuCh^{++}calculus in a similar setting, but it is certainly worthwhile exploring the possible connections between it and Breu's treatment of abstract data types.

Future Work

A complete meta-theoretical study of the CuCh^{++} calculus is sought. At the moment, we are working on proving the correctness of our encoding τ according to the following steps:

- define an evaluation relation $\rightarrow_\Delta$ for a CuCh expression inside an environment Δ, containing algebra definitions and recursive equations, as the reduction relation defined by the call-by-name β-reduction and the rewriting rules obtained by orienting the equations;
- define a set of values both for CuCh^{++} and CuCh, and hence state what termination means in both calculi;
- finally, show that: (i) if Δ is the set of CuCh definitions obtained by translating a set of CuCh^{++} class definitions Γ, the encoding τ commutes with respect to $\rightsquigarrow_\Gamma$ and $\rightarrow_\Delta$ (*soundness*); (b) the evaluation of a CuCh^{++} program does not terminate if and only if the evaluation of the corresponding (obtained via τ) Cuch expression does not terminate either (*adequacy*).

An earlier version of the translation presented in Section 4 was implemented as a "precompiler" which takes as input a CuCh^{++} program and produces the corresponding CuCh code. We would rather not hiding the CuCh programming style but instead extending the CuCh machine with CuCh^{++} object–oriented features in such a way that functional and object–oriented features are blended, in order to get a nice heterogeneous environment to experiment CuCh^{++}'s future extensions. We plan, in fact, to study further extensions of CuCh^{++} and of the encoding, in order to deal with issues such as multiple inheritance and more refined encapsulation rules, to provide the programmer with an explicit control over field and method external accessibility (for example introducing usual keyword **public**, **protected** and **private**).

An important development of the work presented in this paper would be studying an appropriate type system for CuCh^{++}, mainly to be able to statically detect typical run-time errors, such as "message not understood". Some considerations make us slightly optimistic about the design of a type system for CuCh^{++} and also for CuCh (that is untyped), such that our translation

would faithfully map well-typed CuCh^{++} programs into well-typed CuCh ones:

- CuCh algebras suggest themselves a straightforward type system. Instead of specifying only the arity of a constructor, it seems natural to consider typed signature, requiring that the definition of an algebra specifies also the argument types of the constructors.
- Efforts devoted to make the translation deal with problems such as method specialization (at the term level), seems to be of some help for the study of the same problem at the type level.
- Some of the typing problems of interpreting message invocation as function application ([1], chapter 6 and 18), which make faithful encoding of objects into typed lambda–calculi problematic, are easier to manage within an overloading–based model.

As pointed out by an anonymous referee, since we implement an object–oriented language via primitive recursion without using fixed point operators, it might be possible to similarly design a typed CuCh^{++} without recursive types. For example, given a type $Point = \{x : Int,\ move : Int \to Point\}$, can we have a "finite" type for Point? The answer is very likely to be "yes", by using higher–type functionals. It's Böhm-Berarducci [9] all over again, just at the level of types instead of terms.

Acknowledgement

We are grateful to Mariangiola Dezani–Ciancaglini for her careful reading of an earlier manuscript and to the anonymous referees for their suggestions about a possibly typed CuCh^{++}.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.

[2] M. Abadi, L. Cardelli, and R. Viswanathan. An Interpretation of Objects and Object Types. In *Proceedings of 23rd ACM Symposium on Principle of Programming Languages*, pages 396–409, 1996.

[3] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[4] J. Backus. Can Programming be Liberated from vonNeumann Style? A Functional Style and its Algebra of Programs. *ACM Communication*, 21(8):613–641, 1978.

[5] A. Berarducci and C. Böhm. General Recursion on Higher–Order Term Algebras. In *Proceedings of RTA'01*, LNCS. Springer–Verlag, 2001. To appear.

[6] C. Böhm. The CUCH as a Formal and Description Language. In Steel T. B., editor, *Formal Languages Description Languages for Computer for Computer Programming*, pages 179–197. North Holland, 1966.

[7] C. Böhm. Combinatory Foundation of Functional Programming. In *ACM Symposium on Lisp and Functional Programming*, pages 29–36, 1982.

[8] C. Böhm. Functional Programming and Combinatory Algebras (invited lecture). In M.P. Chytil, L. Janiga, and Koubek V., editors, *Proceedings of 4th Mathematical Foundations of Computer Science*, volume 324 of *LNCS*, pages 14–26. Springer–Verlag, 1988.

[9] C. Böhm and A. Berarducci. Authomatic Synthesis of Typed λ–Programs on Term Algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[10] C. Böhm and M. Dezani-Ciancaglini. A CUCH-machine: the Authomatic Treatment of Bound Variables. *International Journal of Computer and Information Sciences*, 1(2):171–191, 1972.

[11] C. Böhm and M. Dezani-Ciancaglini. Notes on a CUCH-machine: the Authomatic Treatment of Bound Variables. *International Journal of Computer and Information Sciences*, 2(2):157–160, 1973.

[12] C. Böhm and W. Gross. Introduction to the CUCH. In E. R. Caianiello, editor, *Automata Theory*, pages 35–65. Academic Press, 1966.

[13] C. Böhm, A. Piperno, and S. Guerrini. Lambda-definition of function(al)s by normal forms. In D. Sannella, editor, *Proceedings of ESOP'94*, volume 788 of *LNCS*, pages 135–149. Springer–Verlag, April 1994.

[14] V. Bono and M. Bugliesi. Interpretations of extensible objects and types. In *FCT'99*, volume 1684 of *LNCS*, pages 112–123. Springer-Verlag, 1999. An extended version was accepted for pubblication on ACM Transaction TOCL, 2001.

[15] V. Bono and K. Fisher. An Imperative First-Order Calculus with Object Extension. In *ECOOP '98*, volume 1445 of *LNCS*, pages 462–497. Springer-Verlag, 1998. (A preliminary version appeared in the electronic proceedings of FOOL5, 1998).

[16] V. Bono, A. Patel, V. Shmatikov, and J. C. Mitchell. A core calculus of classes and objects. In *MFPS'99*, volume 20 of *Electronic Notes in TCS*, pages 1–22. Elsevier, 1999.

[17] R. Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments*, volume 562 of *LNCS*. Springer-Verlag, 1991.

[18] K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing Objects Encondings. *Information and Computation*, 155(1/2):108–133, 1999.

[19] K.B. Bruce, L. Petersen, and A. Fiech. Subtyping is not a Good "Match" for Object-Oriented Languages. In *Proceedings of ECOOP'97: European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 104–127. Springer–Verlag, 1997.

[20] K.B. Bruce, A. Shuett, and R. van Gent. PolyTOIL: a Type-Safe Polymorphic Object-Oriented Language. In *Proceedings of ECOOP'95: European Conference on Object-Oriented Programming*, volume 952 of *LNCS*, pages 27–51. Springer–Verlag, 1995.

[21] L. Cardelli. A Semantics of Multiple Inheritance. *Information and Computation*, 76(2/3):138–164, 1988.

[22] G. Castagna. A Meta Language for Typed Object Oriented Languages. *Theoretical Computer Science*, 151(2):297–352, 1995.

[23] G. Castagna, G. Ghelli, and B. Longo. A Calculus of Overloaded Functions with Subtyping. *Information and Computation*, 117(1):115–135, 1995.

[24] W.R. Cook. *A Denotational Semantics of Inheritance.* PhD thesis, Brown University, 1989.

[25] E. Ellis and B. Stroustrop. *The Annotated C^{++} Reference Manual.* ACM Press, 1990.

[26] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[27] K. Fisher and J. C. Mitchell. On the Relationship between Classes, Objects and Data Abstraction. *Theory and Practice of Object Systems*, 4(1):3–32, 1998.

[28] A. Goldberg and D. Robson. *Smalltalk-80, The Language and its Implementation.* Addison Wesley, 1983.

[29] M. Hoffman and B. C. Pierce. A Unifying Type–Theoretic Framework for Objects. *Journal of Functional Programming*, 5(4):593–635, 1995. A preliminary version appeared in Proceedings of TACS94.

[30] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java - A Minimal Core Calculus for Java and GJ. In *Proc. of OOPSLA*, volume 34 of *ACM SIGPLAN Notices*, pages 132–146. ACM press, 1999.

[31] S.N. Kamin and U.S. Reddy. Two semantic models of object-oriented languages. *C.A.Gunter and J.C.Mitchell (eds) Theoretical Aspects of Object-Oriented Programming*, pages 463–496, 1994.

[32] A. Manieri. Un modello funzionale ad oggetti: Cuch^{++}. Tesi di Laurea, Università di Roma, La Sapienza (in italian), 1998.