*Electronic Notes in Theoretical Computer Science 50 No. 2 (2001) { Proc. BOTH 2001 URL:* [*http://www.elsevier.nl/locate/entcs/volume50.html*](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, Universita degli Studi di Torino corso Svizzera 185, 10149 Torino, Italy,*

*e-mail:* [*fbono,salvog@di.unito.it*](mailto:fbono%2Csalvog@di.unito.it)

*Abstract*

*Bohm's CuCh machine extends the pure lambda{calculus with algebraic data types and provides the possibility of de ning functions over disjoint sums of algebras. We exploit such natural form of overloading to de ne a functional interpretation of a simple, but signi cant fragment of a typical object{oriented class{based language.*

*Dedicated to Professor Corrado Bohm 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 de nition 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 com- ponents 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 Co n '99 TOSCA.*

*c 2001 Published by Elsevier Science B. V. Open access under* [*CC BY-NC-ND license.*](http://creativecommons.org/licenses/by-nc-nd/3.0/)

*inheritance mechanisms. Classes have been studied either as a primitive con-* struct ([20,19,16,30]), or as a combination among extensible objects and tra- ditional 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 di erent translations of object-based calculi into di erent 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 func- tional paradigms, even though object-orientation was born from the impera- tive paradigm. In particular, the functional setting we consider, the Cuch ma- chine [8,13], o ers algebraic data types and the possibility of de ning 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 Bohm'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 embod- ies 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 inter- pret classes as algebras, object constructors as algebraic constructors, objects as terms of such algebras, and methods as recursively de ned functions, and to take advantage of the possibility of de ning 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 elds) are naturally interpreted in the language CuCh.

*In this paper we present the basic CuCh++ ideas and the formal de nition*

*of our CuCh++ {to{Cuch encoding, but we defer to future work its meta-* theoretical analysis. In Section 5 we will brie y 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 Bohm'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 de nition. 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 pro- gramming 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 imper- ative programming languages which are related rather to the von Neumann's architecture than to problem speci cations. 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" de ned on that algebras, using second order typed normal forms, was introduced by Bohm and Berarducci in [9].

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

*More precisely, let be a signature, partitioned into two sets, 0 =* fc1;::: ; clg (data constructors) and 1 = ff1;::: ; fkg (function symbols). Given a set of equations:

*fi(cj x1 ::: xn)y1 ::: ym = bi;j*

*where 1 i k; 1 j l and bi;j an extended lambda-term with symbols* from the signature , it is possible to nd normal solutions to this system, i.e., normal-form lambda-terms that, once substituted for fi and cj, 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 Curry's combinators and Church's lambda terms, whereas the latter allows de nitions of data belonging to arbitrary (free) term algebras, and recursive de nitions of functions whose rst argument belongs to such an algebra. Algebras are de ned specifying the constructors and their arity. Functions are de ned by systems of recursive equations. An interesting aspect is that functions can be de ned 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 de ne the algebras of natural numbers* and (polymorphic) lists 2 and how to de ne an overloaded function sum which

*2 In the latest version of the CuCh machine, natural numbers and lists are built-in data- types. The abbreviations n for succn 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 := fsucc:1, zero:0g; List := fcons:2, nil:0g; 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, nding a map which associates to each symbol sum, cons, nil, succ, zero a lambda-term such that the equations are satis ed.

*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 func- tion de nitions from the usual treatment via xed point combinators: here functions (programs) are normal forms which, when applied to some argu- ments, incidentally may diverge;

*the engine of the recursion is data, so we may talk of data{driven program-* ming, 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 de ne 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 de nitions. We choose not to include classes as rst-order* entities because this makes easier to deal with class names, which, in turn, make easier de ning the encoding of Section 4 where classes are seen as al- gebras (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 de nition speci es the superclass' class, together with the behavior of its objects by listing the elds and the method names (and respective bodies). A class silently inherits all methods and elds 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

*elds nor methods are de ned. The visibility rules for this rst release of the*

*CuCh++ are inspired from a sort of \purist" object{oriented point of view,* in which elds are private (i.e, accessible only via methods de ned inside the class) and methods are public (i.e., accessible everywhere).

*We consider an extended lambda{calculus equipped with a set of constants*

*= 0 [ 1 [ 2, from where we choose respectively class, eld and method* names. In the CuCh++ syntax we distinguish between class declarations, indicated by the symbol D, and expressions, indicated by the symbol e.

*D ::= class C is C0*

*f v1; ::: ; vk;* meth m1 = e1;

*.*

*meth mn = en g*

*where C; C0 2 0, v1;::: ; vk 2 1, and m1;::: ; mn 2 2. The keyword is* introduces the superclass C0 of the class C.

*P ::= let D1 ::: Dn in e n 0* where e is an expression de ned as follows.

*e ::= x variable*

*x:e abstraction*

*(e1e2) application*

*e ( m method invocation*

*new C object creation* e1#v = e2 eld update

*e:v eld selection*

*self*

*super ( m*

*where x is a variable, C 2 0;m 2 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(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.*

*e1#v = e2 denotes the object o where the content of the eld v in o is re-* placed with e2, if e1 reduces to an object o;

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

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

*super allows the previously de ned code of an overridden method to be re-* ferred. super ( m has a static meaning (as it has in the most common object{oriented languages, from Smalltalk [28] to Java [3]), i.e., super ( 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 in uence the binding for self;

*The basic object operations ( eld update and eld selection) can be per-* formed 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: hv1 = e1;::: ; vk = ek; m1 = b1;::: ; mn = bn; parent = ei, where vi's are its elds and mj's are its methods. When k is 0 we have an object without elds and when n is 0 the object has no methods attached. An object generated by the top class Empty has then the shape: hi. Observe that an object is a nested record, since the expression associated to the special eld 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 seman- tics rules. Notice also that a eld 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 de ne inductive data types, like natural numbers or lists, via class de nitions.

*Given the two above mentioned restrictions, we can now de ne what a* legal CuCh++ program is.

*De nition 3.1 [Programs] A declaration let C1 ::: Cn 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 de nitions.

*(ii) objects do not appear in e.*

*3.1 Operational Semantics*

*e ;fD ; :::; D g e0*

*1 n*

*(let)* *0*

*( )*

*(()*

*let D1; ::: ; Dn in e ;? e*

*e1 ; ( x:e0 )*

*1*

*1*

*(e1e2) ; e0 [x := e2]*

*e ; o o = hvi = ei; mj = bj; parent = e0i m = mj e ( m ; (o:m o)*

*0*

*e1 ; o o = hvi = ei; mj = bj; parent = e0i v = vi*

*0*

*(upd1)*

*e #v = e*

*hv = e (i 6= i );v*

*= e ;m*

*= b ; parent = e0i*

*1 2 ; i* *i*

*0 i0*

*2 j j*

*e1 ; o o = hvi = ei; mj = bj; parent = e0i v 6= vi8i*

*(upd2)*

*e #v = e*

*hv = e ;m*

*= b ; parent = (e0#v = e )i*

*1 2 ; i i j j* *2*

*(ext1)*

*e ; o o = hni = ei; parent = e0i n = ni e:n ; ei*

*0*

*0*

*e ; o o = hni = ei; parent = e0i 8i n 6= ni*

*(ext2)*

*e:n ; e0:n*

*class C is C0fv1;::: ; vk; meth m1 = b1 ::: meth mn = bng 2 new C0 ; y1 ::: yl:e0 e00*

*x1;::: ; xk; z1;::: ; zl fresh variables*

*(new1) new C ; x1 ::: xkz1 ::: zl:hvi = xi; mj = bj; parent = (e00 z1 ::: zl)i (new2) new Empty ; hi*

*Fig. 1. Operational Semantics of CuCh++*

*class C is C0fv1;::: ; vk; u1 = e1;::: ; ur = er;*

*meth m1 = b1 ::: meth mn = bng*

*2 new C0 ; y1 ::: yl:e0*

*new C ;*

*x1 ::: xky1 ::: yl:*

*hvi = xi; uh = eh; mj = bj; parent = (e0 y1 ::: yl)i*

*Fig. 2. An alternative rule for new*

*Figure 1 shows the formal de nition of the operational semantics for the* language CuCh++ . The operational reduction relation ; is parameterized

*with respect to an environment , built up from the class declarations (rule* (let)). The environment 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 b0 = self :b[self =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 super ( m with the expression bmself, where bm 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) ( ) rule of lambda{calculus. The method* invocation rule (() looks up for the appropriate method body by triggering the (ext ) rules. Rule (ext1) takes care of the case when a method was newly introduced by the class of the receiver. Rule (ext2 ), instead, looks up for the method in the hierarchy, by re-triggering the search on the parent content, which contains elds 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 (ext1) and (ext2) are also used for the retrieval of a eld value. Rules (upd1), (upd2) deal with the replacement of a eld value and they basically work as the (ext ) rules in order to retrieve the right eld to be updated. In rules (ext ) and (upd ), we use the metavariable n to range over both eld and method names. Rule (new1) shows how objects are instantiated from classes. A new expression evaluates to a function whose parameters, x1;::: ; xk; z1;::: ; zl are the initial values for the elds (both new, the xi's, and inherited, the zj's), so that actual values must be supplied. Thanks to the special eld parent, the initialization phase for the superclass' elds is performed without having to explicitly mention their names. Delegating the superclass to deal with its own elds is good design practice (see [27]), and even though in our case a class must be aware of the number of elds of its immediate superclass, this is the only information leakage. Rule (new2) 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 eld in the class de nition. In such a case the rule (new) should be rewritten as in Figure 2.

# *4 The Translation*

*In this section we de ne our translation of CuCh++ programs into CuCh* programs. First we informally describe some basic ideas, then we present an example, and nally we give the formal de nition of the translation.

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

*The mechanism of the Cuch machine for de ning functions over the union* of algebras allows to associate di erent \pieces of code" to the same func- tion, depending on the data constructor of the rst argument (overloading). Therefore, translation of overriding methods is straightforward thanks to this feature: it suÆces to translate the overriding method body. An inherited method is translated by extending the de nition of the function correspond- ing 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 de nition produces also the de nition 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 de nitions of three classes in CuCh++, Point,* C Point and B C Point, show how these de nitions are translated into Cuch language, and brie y 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 de nitions). 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 pre xed. Moreover, we freely use in both CuCh++ and CuCh examples some syntactic sugar such as if b then e1 else e2 statements (if b e1 e2 in CuCh).

*The de nition of the Point class de nes the behavior of a one-dimensional* point. The eld 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 Emptyfx;* meth get x = self:x

*meth move = d:self#x = (+ self:x d)*

*meth dist = abs self:x*

*g*

*C Point (subclass of Point class) de nes the behavior of a colored point. A*

*eld 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 xed interval.

*class C Point is Point fc;* meth get color = self:c

*meth set color = col:if(< (self ( dist) 3)*

*then self#c = col*

*else self#c = BLACK* g

*Let us now de ne the class of bidimensional point, B C Point, where we over-* ride methods dist and move:

*class B C Point is C Point fy;*

*meth dist = max (abs self:y) (super ( dist)*

*meth move = dxdy:(super ( move dx)#y = (+ self:y dy))*

*g*

*The translation of above de nitions gives rise to the following CuCh de ni-* tions.

*Classes.*

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

*Classes:=fPOINT:0, C POINT:0, B C POINT:0, EMPTY:0g*

*Point:=fpoint:1g*

*C Point:=fc point:2g*

*B C Point:=fb c point:2g Empty:=fempty:0g*

*The algebra Classes has a 0-ary constructor for each class de nition. An* algebra with a set of 0-ary constructors allows the de nition of functions over

*a nite domain and hence di erent equations (representing di erent 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 de nition we generate an algebra de ni-* tion.

*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 elds).

*Field selection and eld update.*

*It turns out that to properly translate eld selection and eld update (#* and : operators) we need to de ne a set of auxiliary functions 4 . For each

*eld v (v) we de ne two Cuch functions, sv and uv, and we translate each* subexpression of the form e1#v = e2 to (uv e1 e2), where e1 (resp. e2) is the translation of e1 (resp. e2). Similarly, we translate subexpression of the form e:v with (sv e) where e is the translation of e. When we de ne a subclass M of a class N, we uniformly specialize the auxiliary functions on the elements of the CM subalgebra simply by adding the corresponding equations, taking advantage of CuCh overloading.

*ux (point x parent) z = point z sx (point x parent) =* *x*

*uc (c point c parent) z = c point z parent sc (c point c parent) = c*

*ux (c point c parent) z = (c point c (ux parent z))*

*sx (c point c parent) = sx parent*

*uy (b c point y parent) z = b c point z parent sy (b c point y parent) = y*

*ux (b c point y parent) z = (b c point y (ux parent z)) sx (b c point y parent) = sx parent*

*uc (b c point y parent) z = (b c point y (uc parent z))*

*sc (b c point y parent) = sc parent*

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

*:*

*when they*

*Methods (newly de ned and overriding).*

*We translate each newly de ned and overriding method body into a func-* tion that takes as its rst parameter an element of the algebra representing the class the method belongs to, this parameter leading the dynamic lookup pro- cess (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 elds, 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 rst 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. sx self*

*move POINT = lmb: self d. ux self (+ (sx self) d) dist POINT = lmb: self. (abs sx self)*

*get color C POINT = lmb: self. sc self*

*set color C POINT = lmb: self c. uc self c move B C POINT = lmb: self dx dy.*

*uy (move C POINT self dx) (+ (sy self) dy) dist B C POINT = lmb: self.*

*max (abs (sy 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 de ne the subclass C Point, we expect that the get x and move* methods, de ned in the class Point, to work properly on C Point objects. To obtain this in the CuCh translation, we have to introduce equations which de-

*ne appropriate functions over the C Point algebra, i.e., equations that extend*

*the get x and move de nitions 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 de nition, and hence de ned 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 de ne new in CuCh as many class de nitions 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 de nitions:

*((new B C Point 5 GREEN 2) ( set color RED) ( get color*

*The evaluation of such expression gives as result BLACK because of dy-* namic method lookup: the invocation of the dist method in the body of set color executes the body of the dist method de ned 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 consid-* ering CuCh equations as a rewriting system.

*Formal De nition of the Translation Expressions*

*(x) = x*

*( x:e) = lmb: x: (e)*

*(e1 e2) = ( (e1) (e2))*

*(e ( m) = m D( (e)) (e)*

*(new M) = new M*

*(e1#v e2) = uv (e1) (e2)*

*(e:v) = sv (e)*

*(self) = self*

*(super ( m) = m CN self*

*Translation of a Class De nition*

*The translation of a CuCh++ class de nition of the shape:*

*class M is N*

*f v1; ::: ; vk;* meth m1 = e1;

*.*

*meth ml = el g*

*corresponds to a sequence of CuCh de nition, as follows:*

*algebra de nitions A de nition of a free algebra with a data constructor of* arity k + 1, a 0-ary constructor CM to be added in the algebra Classes:

*M:=fcM:k+1g Classes:=f..., CM:0g and the auxiliary D function:*

*D (cM v1 ...vk) = CM*

*auxiliary functions For each eld vi the de nition of the auxiliary functions* uvi and svi is as follows:

*uv1 (cM v1 v2 ...vk parent) z = cM z v2 ...vk parent*

*.*

*uvk (cM v1 v2 ...vk parent) z = cM v1 v2 ...z parent sv1 (cM v1 ...vk parent) = v1*

*.*

*svk (cM v1 ...vk parent) = vk*

*and the specialization of the auxiliary functions (de ned in the translation*

*of the parent class N) is as follows:*

*uy1 (cM v1 v2 ...vk parent) z =*

*cM v1 v2 ...vk (uy1 parent z)*

*.*

*uyl (cM v1 v2 ...vk parent) z =*

*cM v1 v2 ...vk (uyl parent z)*

*sy1 (cM v1 ...vk parent) = sy1 parent*

*.*

*syl (cM v1 ...vk parent) = syl parent*

*methods (newly de ned and overriding) For each method mi new or* overriding we have:

*mi CM = lmb: self. (ei)*

*methods (inherited) For all functions ni generated from the encoding of* methods of N which are not overridden in M we add the equation:

*ni CM = ni CN*

*new Let's assume that the objects of N has j elds. The de nition of the* new function is then as follows:

*new CM = lmb: v1...vk y1 ...yj. cM v1...vk (new CN y1...yj)*

# *5 Conclusions*

*We presented an (untyped) object{oriented calculus CuCh++ inspired by the* Bohm's Cuch philosophy, and provided a CuCh++{to{Cuch encoding that takes care of some of the main concepts of object{oriented programming lan- guages, such as classes, objects, inheritance, dynamic method lookup and a primitive form of encapsulation (public methods and private elds). 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: asa 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 xed 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 t 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, &, where overloaded terms and types are considered. In [22], an encoding of a toy object{oriented language into & has been proposed. In the &-calculus, an overloaded term is essentially a collection of functional terms, glued together. When an overloaded term M = M1& ::: &Mn is applied to an argument N, one branch Mi of the over- loaded 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 opera- tional 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 Interpreta- tion 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 speci cation techniques and object{ oriented programming is presented. This work is really interesting since it exploits algebraic data types de nitions 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 connec- tions 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 according to the following steps:

*de ne an evaluation relation ! for a CuCh expression inside an envi-* ronment , containing algebra de nitions and recursive equations, as the reduction relation de ned by the call-by-name -reduction and the rewriting rules obtained by orienting the equations;

*de ne a set of values both for CuCh++ and CuCh, and hence state what* termination means in both calculi;

*nally, show that: (i) if is the set of CuCh de nitions obtained by trans-* lating a set of CuCh++ class de nitions , the encoding commutes with respect to ; and ! (soundness); (b) the evaluation of a CuCh++ pro- gram does not terminate if and only if the evaluation of the corresponding (obtained via ) Cuch expression does not terminate either (adequacy).

*An earlier version of the translation presented in Section 4 was imple-* mented as a \precompiler" which takes as input a CuCh++ program and pro- duces 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 re ned encapsulation rules, to provide the pro- grammer with an explicit control over eld 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 stati- cally 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 de nition of an algebra speci es also the argument types of the constructors.

*E orts 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 func-* tion 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 xed point operators, it might be possible to similarly design a typed CuCh++ without recursive types. For example, given a type P oint = fx : Int; move : Int ! P ointg, can we have a \ nite" type for Point? The answer is very likely to be \yes", by using higher{type functionals. It's Bohm-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. Bohm. General Recursion on Higher{Order Term Algebras. In Proceedings of RTA'01, LNCS. Springer{Verlag, 2001. To appear.*

*[6] C. Bohm. 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. Bohm. Combinatory Foundation of Functional Programming. In ACM Symposium on Lisp and Functional Programming, pages 29{36, 1982.*

*[8] C. Bohm. 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. Bohm and A. Berarducci. Authomatic Synthesis of Typed {Programs on Term Algebras. Theoretical Computer Science, 39:135{154,* *1985.*

*[10] C. Bohm 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. Bohm 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. Bohm and W. Gross. Introduction to the CUCH. In E. R. Caianiello, editor, Automata Theory, pages 35{65. Academic Press, 1966.*

*[13] C. Bohm, A. Piperno, and S. Guerrini. Lambda-de nition 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 Speci cation 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. Ho man 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,*

*Universita di Roma, La Sapienza (in italian), 1998.*