Everything is an Object

Luca Cardelli

based on joint work with Martín Abadi

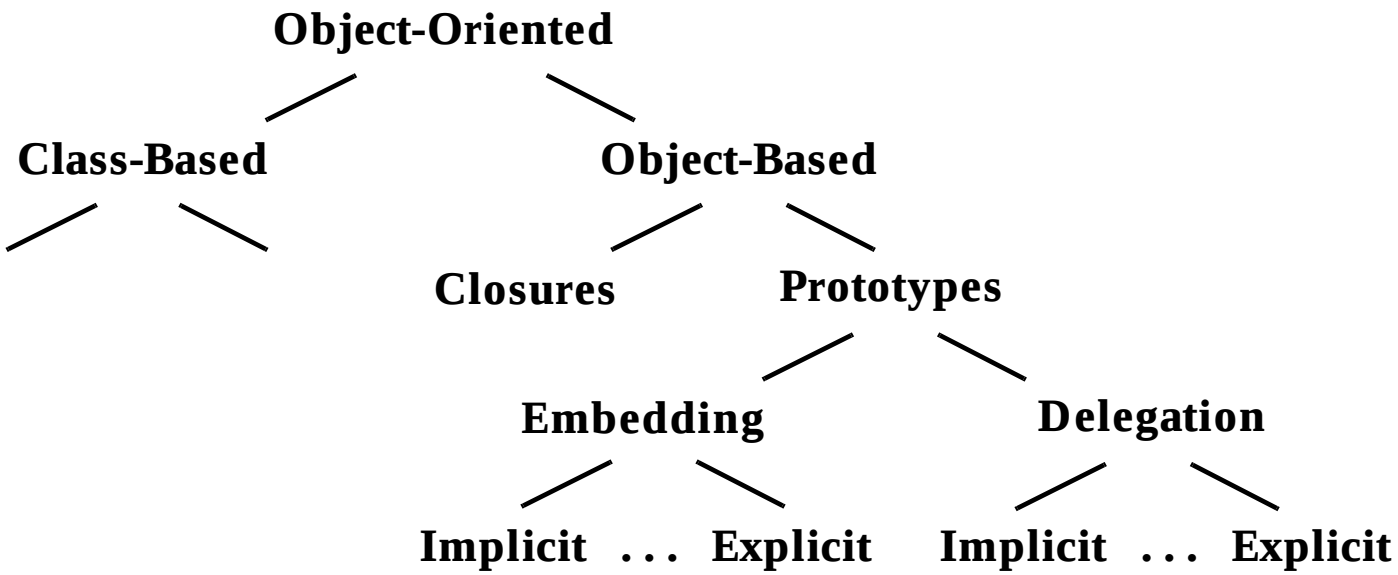
Digital Equipment Corporation

Systems Research Center

Abstract

I discuss the foundations of object-based programming. Recent results validate the long-standing intuition that everything can be represented in terms of objects, including functions and classes. Similarly, function types and class types can be represented via object types. The basic constructions are simple, flexible, and powerful. So, why are object-based languages not taking over the (class-based) world?

A Hierarchy of O-O Features



- I'll try to explain why I think that:
  - ~ Prototype-based languages are not just an obscure sub-sub-branch of a complex language hierachy.
  - ~ They are, foundationally, the most important sub-sub-branch.

- By the eventual inevitability of simplicity:
  - ~ Prototype-based languages should become more prominent.
  - ~ But we do not seem to be there yet.

The Imperative ς-calculus

The "simplest" prototype-based language.

b ::=		terms
	x	identifiers
	$[l_i = \varsigma(x_i)b_i \ ^{i \in 1..n}]$	objects (i.e. **object**[l = **method**()...**self**...**end**, ...])
	$b.l$	method invocation (with no parameters)
	$b_1.l \Leftarrow \varsigma(x)b_2$	method update (imperative)
	$clone(b)$	cloning (shallow copy)
	$let\ x = b_1\ in\ b_2$	local declaration (yields ";" and *fields*)

- Fields can be encoded:

$$[..., l = b, ...] \triangleq let\ x = b\ in\ [..., l = \varsigma(y)x, ...]$$
$$b.l \triangleq let\ x = b\ in\ x.l$$
$$b_1.l := b_2 \triangleq let\ x_1 = b_1\ in\ let\ x_2 = b_2\ in\ x_1.l \Leftarrow \varsigma(y)x_2$$

Basic Examples

Let $o_1 \triangleq [l=\varsigma(x)[]]$ A convergent method.
then $o_1.l \rightsquigarrow []$

Let $o_2 \triangleq [l=\varsigma(x)x.l]$ A divergent method.
then $o_2.l \rightsquigarrow x.l\{x \leftarrow o_2\} \equiv o_2.l \rightsquigarrow \ldots$

Let $o_3 \triangleq [l = \varsigma(x)x]$ A self-returning method.
then $o_3.l \rightsquigarrow x\{x \leftarrow o_3\} \equiv o_3$

Let $o_4 \triangleq [l = \varsigma(y)\ (y.l \Leftarrow \varsigma(x)x)]$ A self-modifying method.
then $o_4.l \rightsquigarrow (o_4.l \Leftarrow \varsigma(x)x) \rightsquigarrow [l = \varsigma(x)x]$

... but also suggestive and expressive.

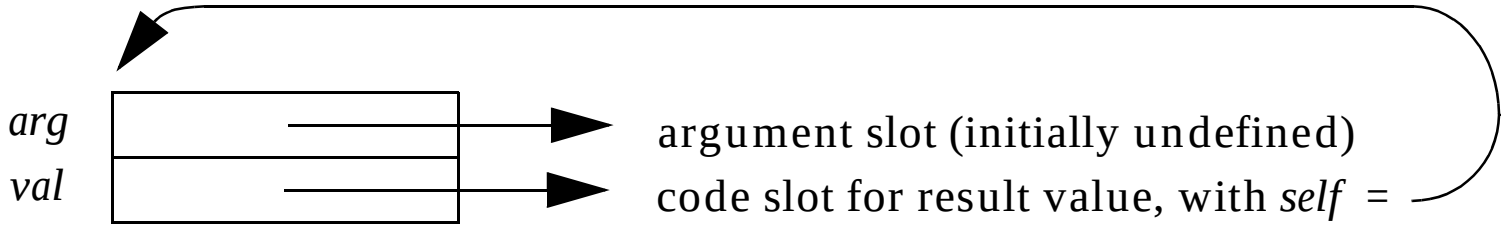
- role of self (hidden recursion)
- data structures (numbers, trees, etc.)
- controls structures (functions, classes, state encapsulation, conditionals, loops, recursion)
- typing (soundness, subtyping, Self types)
- semantics (formal o-o language definitions)

A.k.a. Obliq

b ::=	terms
x	identifiers
$\{l_i$ => **meth**$(x_i)b_i$ **end** $^{i \in 1..n}\}$	objects
$b.l$	method invocation
$b_1.l$:= **meth**$(x)b_2$ **end**	method update
clone(b)	cloning
let $x = b_1$ **in** b_2 **end**	local declaration (yields *fields*)

Stack-Frame Objects

Stack-frame object *f*:



clone(f) is stack-frame allocation (sharing the code for the frame).
f.arg := a is parameter-passing on the stack frame.
f.val is "jumping to the program counter" of the stack frame.

Procedures from Objects

$a, b \;::=\; x \mid x{:=}a \mid \lambda(x)b \mid b(a)$ an imperative λ-calculus

$\langle\!\langle x \rangle\!\rangle \triangleq x$

$\langle\!\langle x := a \rangle\!\rangle \triangleq$

$\quad let\ y = \langle\!\langle a \rangle\!\rangle$

$\quad in\ x.arg := y$

$\langle\!\langle \lambda(x)b \rangle\!\rangle \triangleq$

$\quad [arg = \varsigma(x)\ x.arg,$

$\quad\ val = \varsigma(x)\ \langle\!\langle b \rangle\!\rangle\{\!\{x \leftarrow x.arg\}\!\}]$

$\langle\!\langle b(a) \rangle\!\rangle \triangleq$

$\quad let\ f = clone(\langle\!\langle b \rangle\!\rangle)$

$\quad in\ let\ y = \langle\!\langle a \rangle\!\rangle$

$\quad in\ (f.arg := y).val$

Preview: this translation extends to typed calculi:

$\langle\!\langle A \to B \rangle\!\rangle \triangleq [arg{:}\ \langle\!\langle A \rangle\!\rangle, val{:}\ \langle\!\langle B \rangle\!\rangle]$

- Call-by-value parameter passing is validated:

$$
\begin{aligned}
&\langle\!\langle (\lambda(x)b)(a) \rangle\!\rangle \\
&\quad \equiv \quad \mathit{let}\ f = \mathit{clone}([\mathit{arg} = \varsigma(x)\ x.\mathit{arg},\ \ \mathit{val} = \varsigma(x)\ \langle\!\langle b \rangle\!\rangle\{x \leftarrow x.\mathit{arg}\}]) \\
&\qquad\quad \mathit{in}\ \mathit{let}\ y = \langle\!\langle a \rangle\!\rangle\ \mathit{in}\ (f.\mathit{arg} := y).\mathit{val} \\
&\quad \sim \quad \mathit{let}\ y = \langle\!\langle a \rangle\!\rangle\ \mathit{in}\ [\mathit{arg} = y,\ \ \mathit{val} = \varsigma(x)\ \langle\!\langle b \rangle\!\rangle\{x \leftarrow x.\mathit{arg}\}].\mathit{val} \\
&\quad = \quad \mathit{let}\ y = \langle\!\langle a \rangle\!\rangle\ \mathit{in}\ \langle\!\langle b \rangle\!\rangle\{x \leftarrow y\} \\
&\quad \equiv \quad \mathit{let}\ x = \langle\!\langle a \rangle\!\rangle\ \mathit{in}\ \langle\!\langle b \rangle\!\rangle
\end{aligned}
$$

  (here ~ is equality modulo object identity)

- The technique generalizes easily to multiple parameters, default parameters, and call-by-keyword.
- Thus, procedural languages are reduced to object-oriented languages.

Objects from Procedures

$$\langle\!\langle x \rangle\!\rangle \triangleq x$$

$$\langle\!\langle [l_i = \varsigma(x_i) b_i \ ^{i \in 1..n}] \rangle\!\rangle \triangleq \langle l_i = \lambda(x_i) \langle\!\langle b_i \rangle\!\rangle \ ^{i \in 1..n} \rangle$$

$$\langle\!\langle b.l \rangle\!\rangle \triangleq \langle\!\langle b \rangle\!\rangle.l(\langle\!\langle b \rangle\!\rangle)$$

$$\langle\!\langle b_1.l \Leftarrow \varsigma(x) b_2 \rangle\!\rangle \triangleq \langle\!\langle b_1 \rangle\!\rangle.l := \lambda(x) \langle\!\langle b_2 \rangle\!\rangle$$

$$\langle\!\langle clone(b) \rangle\!\rangle \triangleq clone(\langle\!\langle b \rangle\!\rangle)$$

$$\langle\!\langle let\ x = a\ in\ b \rangle\!\rangle \triangleq let\ x = \langle\!\langle a \rangle\!\rangle\ in\ \langle\!\langle b \rangle\!\rangle$$

(Assuming an encoding of records as procedures, or primitive records.)

Preview: this translation does *not* extend to typed calculi.

$$[l_i{:}B_i \ ^{i \in 1..n}] \triangleq \mu(X)\langle l_i{:}X \to B_i \ ^{i \in 1..n} \rangle$$

But NOT, *e.g.*: $\mu(X)\langle l{:}X \to A,\ l'{:}X \to B \rangle <: \mu(Y)\langle l{:}Y \to A \rangle$

Classes from Objects

- Inheritance is method reuse. One can reuse methods by:
  - ~ sharing them with other objects (*delegation-based*)
  - ~ extracting them from other objects (*embedding-based*)
  - ~ sharing / extracting them from traits or classes (*class-based*)
- Embedding-based inheritance is the simplest.

  But one cannot easily extract a method of an existing object: method extraction is not type-sound in typed languages.
- Delegation-based inheritance is more complex.

  It has been handled formall [Honsell, Fisher, Mitchell], but is harder to think about and to typecheck. We don't discuss it.

- Class-based inheritance is useful or needed anyway.

  We need something like classes, on top of objects, to achieve (typable) inheritance in our object-based framework.

- Here is the general idea:
  - A *pre-method* is a function that is later used (over and over) as a method.
  - A class is a collection of pre-methods plus a way of generating new objects. (I.e., a class is a trait plus a generator.)

Classes and Inheritance

Example

We define classes cp_1 and cp_2 for one-dimensional and two-dimensional points:

$$
\begin{array}{l}
\mathit{let}\ cp_1 = \\
\quad [new = \varsigma(z)[x = \varsigma(s)\ z.x(s),\ mv\_x = \varsigma(s)\ z.mv\_x(s)], \\
\quad\ x = \lambda(s)\ 0, \\
\quad\ mv\_x = \lambda(s)\ \lambda(dx)\ s.x := s.x+dx]; \\
\mathit{let}\ cp_2 = \\
\quad [new = \varsigma(z)[\ldots,\ y = \varsigma(s)\ z.y(s),\ mv\_y = \varsigma(s)\ z.mv\_y(s)], \\
\quad\ x = cp_1.x, \\
\quad\ y = \lambda(s)\ 0, \\
\quad\ mv\_x = cp_1.mv\_x, \\
\quad\ mv\_y = \lambda(s)\ \lambda(dy)\ s.y := s.y+dy]
\end{array}
$$

We define points p_1 and p_2 by generating them from cp_1 and cp_2:

$$
\begin{array}{l}
\mathit{let}\ p_1 = cp_1.new; \\
\mathit{let}\ p_2 = cp_2.new;
\end{array}
$$

Dynamic Inheritance

We change the *mv_x* pre-method of cp_1 so that it does not set the *x* coordinate of a point to a negative number:

$$cp_1.mv\_x \Leftarrow \varsigma(z)\ \lambda(s)\ \lambda(dx)\ s.x := max(s.x+dx,\ 0)$$

- The update is seen by p_1 because p_1 was generated from cp_1.
- The update is seen also by p_2 because p_2 was generated from cp_2 which inherited *mv_x* from cp_1:

$$p_1.mv\_x(-3).x = 0$$
$$p_2.mv\_x(-3).x = 0$$

In General

- If $o \equiv [l_i = \varsigma(x_i) b_i \ ^{i \in 1..n}]$ is an object,

  $c \equiv [new = \varsigma(z)[l_i = \varsigma(s)\ z.l_i(s) \ ^{i \in 1..n}],$
  $l_i = \lambda(x_i) b_i \ ^{i \in 1..n}]$

  then *c* is a class for generating objects like *o*.

- A (sub)class *c'* may inherit pre-methods from *c*:

  $c' \equiv [new = ...$
  $..., l_k = c.l_k, ...]$

- Roughly the same technique extends to various typed calculi.

Typed Classes and Inheritance

If $A \equiv [l_i{:}B_i\ ^{i\in 1..n}]$ is an object type, then:

$$Class(A) \triangleq [new{:}A,\ l_i{:}A{\to}B_i\ ^{i\in 1..n}]$$

where

$new{:}A$ is a **generator** for objects of type A

$l_i{:}A{\to}B_i$ is a **pre-method** for objects of type A

$$c : Class(A) \triangleq$$
$$[new = \varsigma(c{:}Class(A))\ [l_i = \varsigma(x{:}A)\ c.l_i(x)\ ^{i\in 1..n}],$$
$$l_i = \lambda(x_i{:}A)\ b_i\{x_i\}\ ^{i\in 1..n}]$$

We can produce new objects as follows:

$$c.new \equiv [l_i = \varsigma(x{:}A)\ b_i\{x\}\ ^{i\in 1..n}] : A$$

Subsumption Validates Inheritance

Let $A \equiv [l_i{:}B_i\ ^{i\in 1..n}]$ and $A' \equiv [l_i{:}B_i\ ^{i\in 1..n}, l_j{:}B_j\ ^{j\in n+1..m}]$, with $A' <: A$.

Class(A') may inherit from *Class*(A) iff $A'<:A$

Note that *Class*(A) and *Class*(A') are not related by subtyping.

Let c: *Class*(A), then

$c.l_i$: $A \rightarrow B_i <: A' \rightarrow B_i$.

Hence $c.l_i$ is a good pre-method for *Class*(A'). For example, we may define:

$c' \triangleq [new=..., l_i=c.l_i\ ^{i\in 1..n}, ...] : Class(A')$

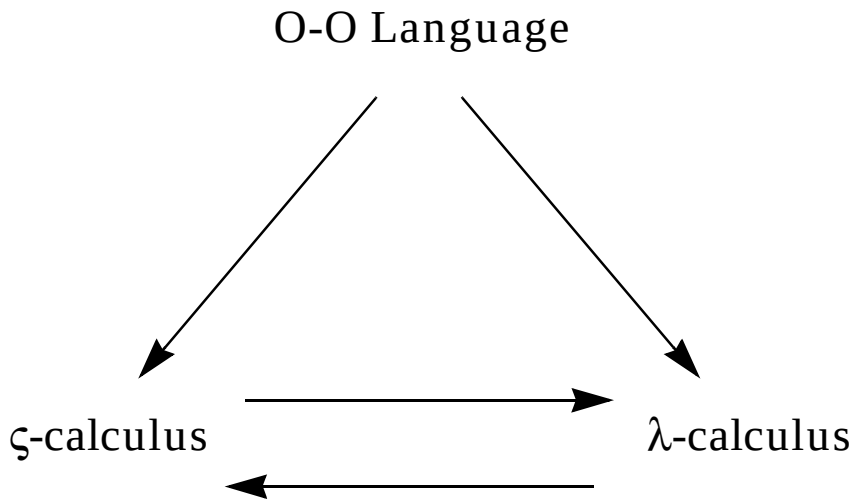
where class c' **inherits** the methods l_i from class c.

TRANSLATIONS

- The representation of object-oriented notions in λ-calculi has normally been carried out informally and incompletely, in terms of examples.
- Object calculi allow us to discuss these representation issues formally and completely, in terms of translations of object calculi into λ-calculi.
- Trying to translate object calculi into λ-calculi means, intuitively, “trying to program in object-oriented style within a procedural language”.

Untyped Translations

- Give insights into the nature of object-oriented computation.
- Objects = records of functions.

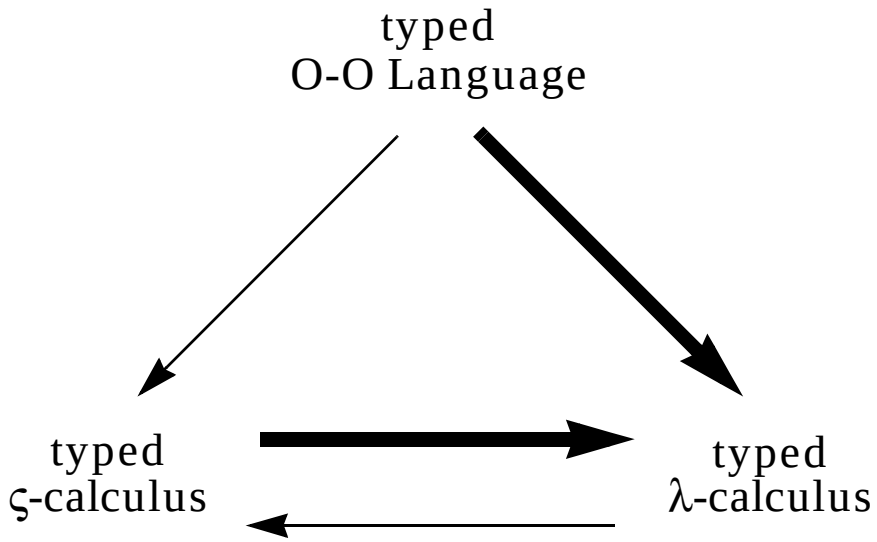


= easy translation

Type-preserving Translations

- Give insights into the nature of object-oriented typing and subsumption/coercion.
- Object types = recursive records-of-functions types.

$$[l_i{:}B_i\ ^{i\in 1..n}] \quad \triangleq \quad \mu(X)\langle l_i{:}X\rightarrow B_i\ ^{i\in 1..n}\rangle$$

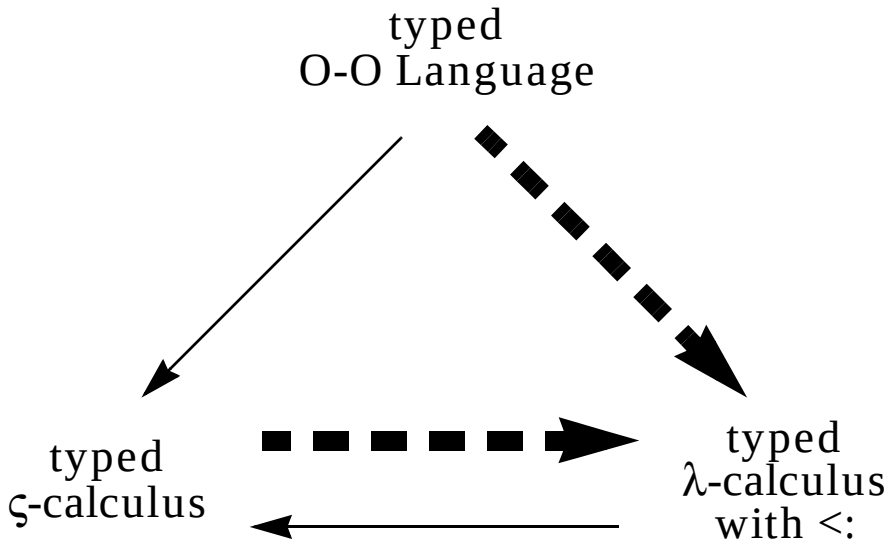


= useful for semantic purposes
impractical for actual programming
losing the “oo-flavor”

Subtype-preserving Translations

- Give insights into the nature of subtyping for object types.
- Object types = recursive bounded existential types.

$$[l_i{:}B_i\ ^{i\in 1..n}] \triangleq \mu(Y)\exists(X<:Y)\langle r{:}X,\ l_i^{sel}{:}X\to B_i\ ^{i\in 1..n},\ l_i^{upd}{:}(X\to B_i)\to X\ ^{i\in 1..n}\rangle$$



= very difficult to obtain,
impossible to use in actual programming

Summary

- *Everything* can indeed be an object. (Even with types.)
  - ~ Objects can emulate procedures (by “stack frame objects”).
  - ~ Objects can emulate classes (by trait-like structures).
  - ~ Objects can also emulate numbers, data structures, etc.
- Conversely, can everything be a function?
  - ~ This is the dominant view in foundations (λ-calculus).
  - ~ Untyped objects can be easily represented functionally.
  - ~ But typed objects are very hard to represent functionally. And even if possible, it is practically unfeasible.
- Hence, objects are more basic than procedures.

Future Directions

- I look forward to the continued development of typed object-based languages.
  - ~ The notion of object type arise more naturally in object-based languages than in class-based languages.
  - ~ Traits, method update, and mode switching are typable (general reparenting is not easily typable).
- No real need for dichotomy.
  - ~ object-based and class-based features can be merged within a single language, based on the common object-based semantics (Beta, O–1, O–2, O–3).

BIBLIOGRAPHY

[1] Abadi, M. and L. Cardelli, **A theory of objects**. Springer. 1996 (to appear).

[2] Abadi, M. and L. Cardelli, **A theory of primitive objects: untyped and first-order systems**. *Proc. Theoretical Aspects of Computer Software*. Lecture Notes in Computer Science 789, 296-320. Springer-Verlag. 1994.

[3] Adams, N. and J. Rees, **Object-oriented programming in Scheme**. *Proc. 1988 ACM Conference on Lisp and Functional Programming*, 277-288. 1988.

[4] Agesen, O., L. Bak, C. Chambers, B.-W. Chang, U. Hölzle, J. Maloney, R.B. Smith, D. Ungar, and M. Wolczko, **The Self 3.0 programmer's reference manual**. Sun Microsystems. 1993.

[5] Alagic, S., R. Sunderraman, and R. Bagai, **Declarative object-oriented programming: inheritance, subtyping, and prototyping**. *Proc. ECOOP'94*. Lecture Notes in Computer Science 821, 236-259. Springer-Verlag. 1994.

[6] Andersen, B., **Ellie: a general, fine-grained, first-class, object-based language**. *Journal of Object Oriented Programming* **5**(2), 35-42. 1992.

[7] Apple, **The NewtonScript programming language**. Apple Computer, Inc. 1993.

[8] Blaschek, G., **Type-safe OOP with prototypes: the concepts of Omega.** *Structured Programming* **12**(12), 1-9. 1991.

[9] Blaschek, G., **Object-oriented programming with prototypes**. Springer-Verlag. 1994.

[10] Borning, A.H., **The programming language aspects of ThingLab, a constraint-oriented simulation laboratory**. *ACM Transactions on Programming Languages and Systems* **3**(4), 353-387. 1981.

[11] Borning, A.H., **Classes versus prototypes in object-oriented languages**. *Proc. ACM/IEEE Fall Joint Computer Conference*, 36-40. 1986.

[12] Cardelli, L., **A language with distributed scope**. *Computing Systems*, **8**(1), 27-59. MIT Press. 1995.

[13] Chambers, C., **The Cecil language specification and rationale**. Technical Report 93-03-05. University of Washington, Dept. of Computer Science and Engineering. 1993.

[14] Chambers, C., D. Ungar, and E. Lee, **An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes**. *Proc. OOPSLA'89*, 49-70. ACM Sigplan Notices 24(10). 1989.

[15] Dony, C., J. Malenfant, and P. Cointe, **Prototype-based languages: from a new taxonomy to constructive proposals and their validation.** P*roc. OOPSLA'92*, 201-217. 1992.

[16] Lieberman, H., **A preview of Act1**. AI Memo No 625. MIT. 1981.

[17] Lieberman, H., **Using prototypical objects to implement shared behavior in object oriented systems**. *Proc. OOPSLA'86*, 214-223. ACM Press. 1986.

[18] Lieberman, H., **Concurrent object-oriented programming in Act 1**. In *Object-oriented concurrent programming*, A. Yonezawa and M. Tokoro, ed., MIT Press. 9-36. 1987.

[19] Madsen, O.L., B. Møller-Pedersen, and K. Nygaard, **Object-oriented programming in the Beta programming language**. Addison-Wesley. 1993.

[20] Paepcke, A., ed., **Object-oriented programming: the CLOS perspective**. MIT Press, 1993.

[21] Rajendra, K.R., E. Tempero, H.M. Levy, A.P. Black, N.C. Hutchinson, and E. Jul, **Emerald: a general-purpose programming language**. *Software Practice and Experience* **21**(1), 91-118. 1991.

[22] Stein, L.A., H. Lieberman, and D. Ungar, **A shared view of sharing: the treaty of Orlando**. In *Object-oriented concepts, applications, and databases*, W. Kim and F. Lochowsky, ed., Addison-Wesley. 31-48. 1988.

[23] Taivalsaari, A., **Kevo, a prototype-based object-oriented language based on concatenation and module operations**. Report LACIR 92-02. University of Victoria. 1992.

[24] Taivalsaari, A., **A critical view of inheritance and reusability in object-oriented programming**. Jyväskylä Studies in computer science, economics and statistics No.23, A. Salminen, ed., University of Jyväskylä. 1993.

[25] Taivalsaari, A., **Object-oriented programming with modes**. *Journal of Object Oriented Programming* **6**(3), 25-32. 1993.

[26] Ungar, D., C. Chambers, B.-W. Chang, and U. Hölzle, **Organizing programs without classes**. *Lisp and Symbolic Computation* **4**(3), 223-242. 1991.

[27] Ungar, D. and R.B. Smith, **Self: the power of simplicity**. *Lisp and Symbolic Computation* **4**(3), 187-205. 1991.

References

http://www.research.digital.com/SRC/personal/Luca_Cardelli/TheoryOfObjects.html