

# On the Semantic Diversity of Delegation-Based Programming Languages

J. Malenfant\*

Département d'informatique et recherche opérationnelle, Université de Montréal, Montréal, Québec, CANADA

## Abstract

The prototype-based programming model has always been difficult to characterize precisely. Its basic principle advocates concrete objects as the only mean to model concepts, yet current languages promote methodologies reintroducing abstract constructions to manage efficiently groups of similar objects. In this paper, we propose a rational reconstruction of delegation-based programming languages that identifies programming models going beyond traditional prototypes. We also introduce a new classification of delegation-based languages, which clarifies these models, and we discuss their relative merits. We finally bring to the fore the existence of more and more structured delegation-based languages forming a continuum between pure prototype-based languages and class-based ones.

## 1 Introduction

Prototype-based programming languages have been proposed more than a decade ago, yet it has been quite difficult to clearly characterize their exact programming model. Part of the difficulty can be attributed to the different semantics they were attributing to the same primitive operations, a problem we dealt with in [5]. But another important source of confusion has been several programming techniques and mechanisms invented to manage groups of similar objects, either in their behavior (e.g. Self's traits objects) or in their representation (e.g. Self's maps). A traits object is a repository for meth-

ods (and "semi-global" variables) applying to the whole group of its delegating objects. A map is a descriptor that factors structural information out of objects in a clone family, i.e. a group of structurally identical objects obtained by cloning one another.

Traits and maps are clearly going against the very notion of prototype-based programming as it has been originally defined, for example by Lieberman [9]. Traits introduce a kind of abstract object while prototype-based languages advocate a programming style based solely on concrete ones. Maps introduce structural descriptors akin to classes, while prototype-based languages were an attempt to build languages around standalone objects only. If indeed traits and maps are alien to *pure* prototype-based programming, from them emerge a much richer notion of *delegation-based programming*. In fact, we show that delegation-based languages exhibit much more diversity than it first appeared, because these new mechanisms impose slightly different programming models, which can hardly be put under the same "prototype-based" hat.

This paper proposes a new classification of delegation-based programming languages, a class of programming languages defined by Wegner [19]. Our new classification completes the one proposed by Wegner, but also the one presented in the Treaty of Orlando [15] as well as our previous one [5] dealing with the primitives of prototype-based programming. We classify delegation-based programming languages according to the number of different kinds of objects and the number of different kinds of links they manipulate. For example, the parent-of link of delegation is one link manipulated in all delegation-based programming languages. Similarly, a trait-based programming language manipulates two kinds of objects: concrete ones and traits. We propose that the number of kinds of links and objects manipulated in a language bears important insights into the nature of its programming model. We explore four classes of delegations-based languages: languages with one kind of object and one kind of link, ones with two kinds of objects and one kind of link, ones with two kinds of objects and two kinds of links and finally ones with one kind of

\*DIRO, Université de Montréal, C.P. 6128, Succursale Centre-ville, Montréal, Québec, Canada H3C 3J7, phone: (514) 343-7479, fax: (514) 343-5834, e-mail : malenfant@iro.umontreal.ca This research has been supported by FCAR-Québec and NSERC-Canada.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA '95 Austin, TX, USA  
© 1995 ACM 0-89791-703-0/95/0010...\$3.50

object and two kinds of links.

But, what constitutes a new kind of object, and what exactly constitutes a link? To answer these questions, our approach studies both the programming methodology and the semantics of four specific languages: a prototype-based language in the line of Lieberman's first proposal, a trait-based language, a map-based language and a descriptor-based language. Our observations allow us to point out exactly where in the semantics of a language differences appear (or do not appear) between subsets of objects. They also allow us to make clear the participation of a link to the semantics of the language.

Prototype-based languages have always been criticized for their lack of manifest organization in programs. Our new classification brings to the fore the existence of delegation-based languages with a more and more structured programming model, forming a continuum between pure prototype-based languages and class-based ones. Not only this shows that the organization of a program can be made more explicit without sacrificing an object-centered programming model, it also suggests a step by step (possibly automatic) transformation of prototype-based programs into class-based ones, a programming methodology advocated before [19, 15].

The rest of the paper is organized as follows. In Section 2, we come back in more details on the prototype-based programming model and justify the introduction of a new terminology, namely object-centered programming, to capture the essence of delegation-based languages. In Section 3, we introduce the basic category of languages with one kind of object and one kind of links, for which we propose a language based on Lieberman's basic assumptions. In this section, we develop the complete syntax and semantics of this language, which will serve as substrate to derive the three other languages. In Sections 4, 5 and 6, we introduce the three other classes in our classification and develop the semantics of three typical languages illustrating each of them. We summarize the classification in Section 7 and we finally close on conclusions and future work.

## 2 Object-centered programming

Prototype-based programming [1, 18, 9, 8, 15, 13] puts forward the fundamental principle that people's natural way to grasp new concepts is generally to begin by creating concrete examples (objects) rather than abstract descriptions (classes). This philosophical statement led to languages that abandon the traditional view of object-oriented programming, namely class-based languages, in favor of an object-centered model. Prototypes are objects that exist on their own, without classes to describe and create them. They are collections of slots representing both instance variables and methods.

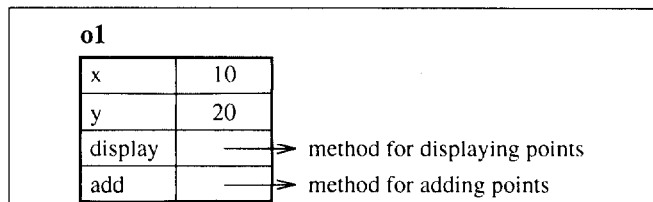


Figure 1: A point example.

For example, in Figure 1, a point object **o1** has four slots named **x**, **y**, **display** and **add**. The two first slots contain values, and their activation by a message simply returns that value, while the two last ones contain methods, and their activation executes the corresponding method. The chief although not unique way to create new prototypes is *cloning*, i.e. shallow copying an existing prototype. Cloning has the side effect that, as long as the two clones don't modify the value of their slots, they will share these values. In [5], we have called this form of sharing *creation-time sharing*, since it lasts for a slot from the cloning until one of the clones changes its value for this slot.

But there is more to prototype-based languages than solely concrete objects. Inheritance is also traded for delegation, a mechanism by which an object that cannot answer a message can delegate it to its parent. Delegation is to concrete objects what inheritance is to classes: a mechanism for sharing information. For example, consider the objects **Clyde** and **Fred** in Figure 2. If a message **legs** is sent to **Fred**, no corresponding slot is found in this object, which therefore delegates the message to **Clyde**. As with inheritance, delegation does not change the nature of the pseudo-variable **self**, i.e. the receiver of the message. When sending a message to **Fred**, a method found in **Clyde** is applied in the context of the receiver **Fred**. In Lieberman's original proposal, delegation is used to make **Clyde** act as *prototypical instance* of elephant: using this prototypical instance, we can create **Fred** differentially by including only the slots that differ from the prototypical instance and use delegation to share **Clyde**'s default characteristics.

The prototype-based model is dominated by the properties of delegation. It is tremendously important to understand that with delegation, we no longer share descriptions, as with class inheritance, but rather concrete representations, and so values of slots. In our previous example, if the state of **Clyde** is modified, so is the state of **Fred**, simply because **Fred** shares with **Clyde** (and with any other object delegating directly or indirectly to **Clyde**), the values of **Clyde**'s slots. Because this form of sharing lasts as long as the two objects exist (if the delegation link cannot be modified), we have called it *life-time sharing*.

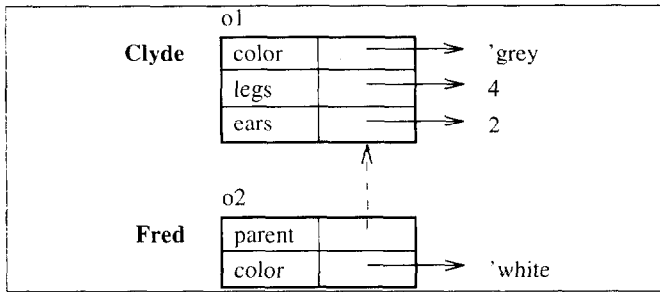


Figure 2: Clyde and Fred.

### Diversity in delegation-based languages

Another important aspect of prototype-based languages that deserves a more complete treatment is the lack of any notion of group of objects. Because there are no classes, we cannot speak about the instances of a class, so there is no clear notion of similar objects, either by their behavior (responding to the same messages) or by their structure (having the same slots). All objects in prototype-based programming are one-of-a-kind. This is the direct consequence of its basic principle, which favors concrete objects over abstractions. But obviously, when similar objects are created, this lack of organization prevents one from making assumptions about them, such as their minimal protocol (something abstract classes nicely give in class-based programming). It also inhibits the efficient representation of families of objects with identical structure, such as the ones created by repeatedly cloning a single prototypical object.

It's no surprise that facing such needs, mechanisms have been proposed to fill the gap. Self, the prototype-based language that has attracted the largest implementation efforts to date, tackled these issues by inventing traits and maps, as we pointed out earlier. Such constructions are a threat to traditional prototype-based programming because they reintroduce abstractness by the back door. A first reaction could be to say that if abstractness is needed, why not going back to class-based languages? Unfortunately, there is something no class-based language offers that delegation-based programming do, namely the value sharing among concrete objects. Even though they will not replace class-based languages as mainstream tools in object-oriented programming, delegation-based languages are a valuable addition to the field and have an interesting application niche (interaction languages, user interfaces, etc.). We take the point of view that concrete objects with delegation are the salient features much sought-after in delegation-based programming languages, and therefore we propose a more general notion of object-centered programming to be contrasted with traditional class-centered programming:

**object-centered programming:** a programming paradigm where the main activity during program design revolves around the creation of concrete objects.

**class-centered programming:** a programming paradigm where the main activity during program design revolves around the creation of classes.

The important point here is that designing an object-centered program really implies the creation of concrete objects as the major activity. In this sense, abstractions like traits and maps are to object-centered programming what assignments are to impure functional programming languages like Scheme and ML: an orthogonal feature which nevertheless leaves the programming model dominated by the major paradigm. As Scheme and ML are dominated by their pure functional subset, delegation-based languages must be dominated by their object-centered one. When introducing abstractions in a delegation-based programming language, the designer must be careful not to encourage (or worse, force) programmers to center their design efforts around them, therefore switching to an abstraction-centered programming model (of which class-based programming is simply an instance). We will see below that several strategies can be applied to ensure this: not adding a sharing mechanism among abstractions (no inheritance between maps for example), preventing abstract objects from receiving messages or, creating them automatically. Keeping these restrictions in mind, it becomes possible to introduce more structure in delegation-based programming languages without sacrificing their object-centered programming model.

### 3 Languages based on one kind of object and one kind of link

Wegner [19] divides object-centered programming languages into two classes: object-based languages, corresponding to one kind of object and no link, and delegation-based languages, i.e. "*classless objects with delegation*", the quintessence of languages with one kind of object and one kind of link (the *parent-of* link). Wegner's classification underestimates the diversity of object-centered programming languages because it is an attempt to characterize the whole space of object-oriented language design. Moreover, at the time of his writing, delegation-based languages were still underinvestigated. In this sense, we complete his classification, being more precise in one of his original class.

Delegation-based languages where the space of objects is completely homogeneous and where delegation is used for sharing, correspond to the usual notion of

prototype-based languages. All objects are equally first-class entities: they can be created dynamically, they can be sent a message, they are all mutable, they can be passed as parameters and returned as results. All of them can be used as parent and cloned. We categorize these languages as having one kind of object and one kind of link, namely the parent-of link.

We illustrate this first category using a minimal language called LIEBERMAN, based on Lieberman's original proposal [9], whose (abstract) syntax appears in Figure 4. This language has standard expressions such as constants (truth values, numerals, characters, strings or symbols), identifiers (accessing the value of let-bound variables), let-expressions, set-expressions (to mutate let-bound variables) and if-expressions for alternatives. Concerning object-centered programming, message passing is accomplished by send-expressions (`send  $\epsilon$   $m$   $e^*$` ) which sends the message  $m$  to the object denoted by  $\epsilon$  with arguments denoted by the expressions list  $e^*$ . We also have self- and super-expressions, which correspond to traditional message sending to self and super. Creation of objects is accomplished using a clone-expression (`clone  $e$` ), which shallow copies the object denoted by  $\epsilon$  and returns the new object as result. An alternative way to create objects is the newInitials-expression (`newInitials  $e$  ( $m^*$ ) ( $e^*$ )`), which creates a new object with slot names denoted by  $m^*$  and slot values denoted by the expressions  $e^*$ ; this new object, which is returned as the result of the expression, will have the object denoted by  $\epsilon$  as parent. The root-expression (`root`) returns as result the object root, the first object in the system which serves as the root of the delegation tree. Finally, we have method-expressions, akin to lambda-expressions, to create methods.

### Wrapper semantics

In the following, we propose a denotational semantics for this minimal language. For that matter, we use the wrapper semantics introduced by Cook and Palsberg [4], which models objects using fixpoints. Let an object  $\mathbf{o1}$  be a function from selectors to values (to be defined later), whose semantic domain is defined as **Object**. The idea is to obtain  $\mathbf{o1}$  by taking the fixpoint of an *object generator*, a function of the form  $(\lambda self \dots)$ , whose domain is **Object**  $\rightarrow$  **Object**. When inheritance or delegation comes into play, the child object is modeled as a function of the form  $(\lambda self \lambda super \dots)$ , where the *super* argument corresponds to the parent object. Such a function is called the *object wrapper*, whose domain is **Object**  $\rightarrow$  **Object**  $\rightarrow$  **Object** (Figure 4 summarizes the semantic domains for objects).

*Wrapper application* creates an object  $\mathbf{o1}$  with parent  $\mathbf{o2}$ . The wrapper of  $\mathbf{o1}$  is applied to the generator of  $\mathbf{o2}$

$\langle \dots \rangle$	list formation
$l \downarrow k$	$k$ th member of the list $l$ (1-based)
$\#l$	length of list $l$
$l \S t$	concatenation of lists $l$ and $t$
$l \dagger k$	drop the first $k$ members of list $l$
$x \downarrow_{\mathbf{D}}$	projection of the value $x$ of some sum domain into the summand $\mathbf{D}$
$in\mathbf{D}(x)$	injection of the value $x$ in the sum domain $\mathbf{D}$
$on_i(x)$	projection of the value $x$ of some product domain onto its $i$ th component

Figure 3: Summary of notation.

by first distributing a new self to the wrapper and the generator and passing the bound generator to the wrapper as its super. The two resulting components are then combined in such a way that a message not answered by  $\mathbf{o1}$ 's now ground wrapper will be forwarded to  $\mathbf{o2}$ 's ground generator. Wrapper application is defined by the operation  $\triangleright$  taking a wrapper and a generator and returning a generator, while left-biased object combination is defined by the operation  $\odot$ :

$$\triangleright = \lambda(\Omega, \Gamma). \lambda\phi. \text{let } o = \Gamma(\phi) \text{ in } \Omega(\phi)(o) \odot o$$

$$\odot =$$

$$\lambda(o, o_1). \lambda m. \text{let } tmp = o(m) \text{ in if } is\mathbf{O}?(tmp) \text{ then } o_1(m) \text{ else } tmp \text{ endif}$$

For more information about wrapper semantics, the curious reader can refer to [4, 7].

### Denotational semantics of LIEBERMAN

In its broad lines, our semantics is written in direct style and it is built around a few important semantic domains. In the following we assume a familiarity with the basic concepts of object oriented programming and some exposure to denotational semantics. We have tried to focus on semantic issues relevant to our classification, deferring technical details that can be skipped over at first reading to the appendix. Valuation functions for expressions (see Fig. 5) take an expression, a self and a super denoting the current object context, an environment for let-bound variables and method formal parameters, an object memory which stores what we call memory objects and methods, and finally a store for other storable values. The store and environment algebras are pretty standard (see [12]). Self and super are values of the domain **Object** introduced before. The last unusual semantic domain is the object memory.

### Syntactic Domains:

$k \in \text{Constants}$   
 $m \in \text{Selectors}$   
 $\nu \in \text{Identifiers}$   
 $e \in \text{Expressions}$   
 $e ::= k \mid \nu \mid (\text{send } e \ m \ e^*) \mid (\text{self } m \ e^*) \mid (\text{super } m \ e^*) \mid (\text{method } (\nu^*) \ e^*) \mid (\text{clone } e) \mid$   
 $(\text{newInitials } e \ (m^*) \ (e^*)) \mid (\text{let } (\nu \ c) \ c^*) \mid (\text{if } e \ e \ e) \mid (\text{set! } \nu \ e) \mid (\text{root})$   
 $k ::= \#t \mid \#f \mid \text{Num} \mid \text{Char} \mid \text{String} \mid \text{Sym}$

### Semantic domains:

Domains for literals and characteristic semantic domains:

$\tau \in \mathbf{T} = \{\perp_{\mathbf{T}}, \text{true}, \text{false}\}$   
 $n \in \mathbf{Num} = \text{unspecified}$   
 $c \in \mathbf{Char} = \text{unspecified}$   
 $s \in \mathbf{String} = \text{unspecified}$   
 $q \in \mathbf{Sym} = \text{unspecified}$   
 $v \in \mathbf{V} = \mathbf{T} \oplus \mathbf{Num} \oplus \mathbf{Char} \oplus \mathbf{String} \oplus \mathbf{Sym}$   
 $\vartheta \in \mathbf{RV} = \mathbf{V} \oplus \mathbf{Oop}$   
 $l \in \mathbf{LV} = \mathbf{Loc}$   
 $\delta \in \mathbf{DV} = \mathbf{EV} = \mathbf{PV} = \mathbf{SV} = \mathbf{RV} \oplus \mathbf{LV}$

Semantic domains for environments and stores:

$t \in \mathbf{O} = \{\perp_{\mathbf{O}}, \top\}$   
 $\nu \in \mathbf{Ide} = \text{unspecified}$   
 $p \in \mathbf{Pointer} = \text{unspecified}$   
 $l \in \mathbf{Loc} = \mathbf{Pointer} \oplus \mathbf{O}$   
 $\rho \in \mathbf{Env} = \mathbf{Ide} \rightarrow (\mathbf{Loc} \oplus \mathbf{T})$   
 $\sigma \in \mathbf{S} = \mathbf{Loc} \rightarrow (\mathbf{SV} \oplus \mathbf{T})$

Semantic domains for objects:

$m \in \mathbf{Selector} = \mathbf{Sym}$   
 $\Delta \in \mathbf{Dict} = \mathbf{Sym} \rightarrow (\mathbf{Loc} \oplus \mathbf{T})$   
 $f \in \mathbf{Fun} = (\mathbf{Om} \otimes \mathbf{S}) \rightarrow \mathbf{PV}^* \rightarrow (\mathbf{EV} \otimes \mathbf{Om} \otimes \mathbf{S})$   
 $\gamma \in \mathbf{Meth} = (\mathbf{Object} \otimes \mathbf{Object}) \rightarrow \mathbf{Fun}$   
 $o \in \mathbf{Object} = \mathbf{Selector} \rightarrow ((\mathbf{T} \rightarrow \mathbf{Fun}) + \mathbf{O})$   
 $\Gamma \in \mathbf{Generator} = \mathbf{Object} \rightarrow \mathbf{Object}$   
 $\Omega \in \mathbf{Wrapper} = \mathbf{Object} \rightarrow \mathbf{Object} \rightarrow \mathbf{Object}$

Semantic domains for the object memory:

$\varrho \in \mathbf{Oop} = \mathbf{Pointer} \oplus \mathbf{O}$   
 $mo \in \mathbf{MObject} = \mathbf{Object} \times \mathbf{Generator} \times \mathbf{Sym}^* \times \mathbf{Loc}^* \times \mathbf{Generator}$   
 $\alpha \in \mathbf{OmV} = \mathbf{MObject} \oplus \mathbf{Meth}$   
 $\mu \in \mathbf{Om} = \mathbf{Oop} \rightarrow (\mathbf{OmV} \oplus \mathbf{T})$

Figure 4: Syntactic and semantic domains for the language LIEBERMAN.

In order to share them among other objects, but also to implement object identities, objects are stored in an object memory, which is a function from object identifiers whose domain is **Oop**, into object memory values, whose domain is **OmV**. The domain of object memory is **Om** (see Section D). Because we also want to be

able to share methods among objects, the object memory contains not only memory objects, whose domain is **MObject**, but also methods, whose domain is **Meth**. Hence, object memory values are in the sum domain **OmV** = **MObject**  $\oplus$  **Meth**. Notice that we have divided storable values among the object memory and the

$\mathcal{L} :: \text{Constants} \rightarrow \mathbf{V}$ $\mathcal{L}[k] = \text{unspecified}$
$\mathcal{E} :: \text{Expressions} \rightarrow (\text{Object} \odot \text{Object}) \rightarrow (\text{Env} \odot \text{Om} \odot \text{S}) \rightarrow (\text{EV} \odot \text{Om} \odot \text{S})$ $\mathcal{E}[k] = \lambda(\phi, \varphi). \lambda(\rho, \mu, \sigma). (\text{inEV}(\text{inRV}(\mathcal{L}[k])), \mu, \sigma)$
$\mathcal{E}[\nu] = \lambda(\phi, \varphi). \lambda(\rho, \mu, \sigma). (\text{inEV}(\text{inLV}(\rho(\nu)  _{\text{Loc}})), \mu, \sigma)$
$\mathcal{E}[(\text{send } \epsilon \ m \ e^*)] =$ $\lambda(\phi, \varphi). \lambda(\rho, \mu, \sigma). \text{let}^* (\epsilon^*, \mu_1, \sigma_1) = \mathcal{RL}[\text{permute}(\langle e \rangle \& \epsilon^*)](\phi, \varphi)(\rho, \mu, \sigma)$ $\text{and } (\epsilon, \epsilon_1^*) = \text{split}(\text{unpermute}(\epsilon^*))$ $\text{in if } \mathcal{L}[m]  _{\text{Sym}=\text{Sym}} \text{ 'set}$ $\text{then } \text{on}_1(\text{bound-object}(\epsilon  _{\text{RV} \text{Oop}})(\mu_1)  _{\text{MObject}})(\epsilon_1^* \downarrow_1  _{\text{RV} \text{V}}  _{\text{Sym}})  _{(\text{T} \rightarrow \text{Fun})} (\text{false})(\mu_1, \sigma_1)(\epsilon_1^* \uparrow 1)$ $\text{else } \text{on}_1(\text{bound-object}(\epsilon  _{\text{RV} \text{Oop}})(\mu_1)  _{\text{MObject}})(\mathcal{L}[m]  _{\text{Sym}})  _{(\text{T} \rightarrow \text{Fun})} (\text{true})(\mu_1, \sigma_1)(\epsilon_1^*)$ $\text{endif}$
$\mathcal{E}[(\text{self } m \ e^*)] = \lambda(\phi, \varphi). \lambda(\rho, \mu, \sigma). \text{let}^* (\epsilon^*, \mu_1, \sigma_1) = \mathcal{RL}[\text{permute}(\epsilon^*)](\phi, \varphi)(\rho, \mu, \sigma)$ $\text{and } \epsilon_1^* = \text{unpermute}(\epsilon^*)$ $\text{in if } \mathcal{L}[m]  _{\text{Sym}=\text{Sym}} \text{ 'set}$ $\text{then } \phi(\epsilon_1^* \downarrow_1  _{\text{RV} \text{V}}  _{\text{Sym}})  _{(\text{T} \rightarrow \text{Fun})} (\text{false})(\mu_1, \sigma_1)(\epsilon_1^* \uparrow 1)$ $\text{else } \phi(\mathcal{L}[m]  _{\text{Sym}})  _{(\text{T} \rightarrow \text{Fun})} (\text{true})(\mu_1, \sigma_1)(\epsilon_1^*)$ $\text{endif}$
$\mathcal{E}[(\text{super } m \ e^*)] = \lambda(\phi, \varphi). \lambda(\rho, \mu, \sigma). \text{let}^* (\epsilon^*, \mu_1, \sigma_1) = \mathcal{RL}[\text{permute}(\epsilon^*)](\phi, \varphi)(\rho, \mu, \sigma)$ $\text{and } \epsilon_1^* = \text{unpermute}(\epsilon^*)$ $\text{in if } \mathcal{L}[m]  _{\text{Sym}=\text{Sym}} \text{ 'set}$ $\text{then } \varphi(\epsilon_1^* \downarrow_1  _{\text{RV} \text{V}}  _{\text{Sym}})  _{(\text{T} \rightarrow \text{Fun})} (\text{false})(\mu_1, \sigma_1)(\epsilon_1^* \uparrow 1)$ $\text{else } \varphi(\mathcal{L}[m]  _{\text{Sym}})  _{(\text{T} \rightarrow \text{Fun})} (\text{true})(\mu_1, \sigma_1)(\epsilon_1^*)$ $\text{endif}$
$\mathcal{E}[(\text{method } (\nu^*) \ e^*)] = \lambda(\phi, \varphi). \lambda(\rho, \mu, \sigma). \text{let } (\epsilon, \mu_1) = \text{make-meth}(\nu^*, \epsilon^*, \mu) \text{ in } (\epsilon, \mu_1, \sigma)$
$\mathcal{E}[(\text{newInitials } \epsilon \ (m^*) \ (\epsilon^*))] = \lambda(\phi, \varphi). \lambda(\rho, \mu, \sigma). \text{let}^* (\epsilon^*, \mu_1, \sigma_1) = \mathcal{RL}[\text{permute}(\langle \epsilon \rangle \& \epsilon^*)](\phi, \varphi)(\rho, \mu, \sigma)$ $\text{and } (\epsilon, \epsilon_1^*) = \text{split}(\text{unpermute}(\epsilon^*))$ $\text{in } \text{allocate-new-object}(\epsilon, \mathcal{LL}[m^*], \epsilon_1^*, \mu_1, \sigma_1)$
$\mathcal{E}[(\text{let } (\nu \ e) \ e^*)] = \lambda(\phi, \varphi). \lambda(\rho, \mu, \sigma). \text{let}^* (\epsilon, \mu_1, \sigma_1) = \mathcal{R}[e](\phi, \varphi)(\rho, \mu, \sigma)$ $\text{and } (l, \sigma_2) = \text{allocation}(\sigma_1)$ $\text{and } \rho_1 = \text{overlay}(\text{binding}(\nu, l), \rho)$ $\text{and } \sigma_3 = \text{store}(l, \epsilon, \sigma_2)$ $\text{in } \mathcal{E}^*[\epsilon^*](\phi, \varphi)(\rho_1, \mu_1, \sigma_3)$
$\mathcal{E}[(\text{if } e \ e_1 \ e_2)] = \lambda(\phi, \varphi). \lambda(\rho, \mu, \sigma). \text{let } (\epsilon, \mu_1, \sigma_1) = \mathcal{R}[e](\phi, \varphi)(\rho, \mu, \sigma)$ $\text{in if } \epsilon  _{\text{RV} \text{V}}  _{\text{T}} \text{ then } \mathcal{E}[e_1](\phi, \varphi)(\rho, \mu_1, \sigma_1) \text{ else } \mathcal{E}[e_2](\phi, \varphi)(\rho, \mu_1, \sigma_1) \text{ endif}$
$\mathcal{E}[(\text{clone } e)] = \lambda(\phi, \varphi). \lambda(\rho, \mu, \sigma). \text{let } (\epsilon, \mu_1, \sigma_1) = \mathcal{R}[e](\phi, \varphi)(\rho, \mu, \sigma) \text{ in } \text{clone}(\epsilon, \mu_1, \sigma_1)$
$\mathcal{E}[(\text{set! } \nu \ e)] = \lambda(\phi, \varphi). \lambda(\rho, \mu, \sigma). \text{let } (\epsilon, \mu_1, \sigma_1) = \mathcal{R}[e](\phi, \varphi)(\rho, \mu, \sigma) \text{ in } (\text{unspecified}, \mu_1, \text{store}(\rho(\nu)  _{\text{Loc}}, \epsilon, \sigma_1))$
$\mathcal{E}[(\text{root})] = \lambda(\phi, \varphi). \lambda(\rho, \mu, \sigma). (\text{inEV}(\text{inRV}(\text{root-oop})), \mu, \sigma)$

Figure 5: Valuation functions for the language LIEBERMAN. (see also Figure 10)

store in order to clearly distinguish object identifiers, which are (storable) r-values, from locations which are l-values. This is an important distinction often skipped over.

Consider now the denotational model of objects. Besides the wrapper semantics, we need a denotational model of the object slots. Slot values are allocated in the store, and the mapping from slot names to the corre-

sponding locations in the store is made by a *dictionary*, i.e. a function from symbols (slot names) to locations (see Section A.1). A slot value can be either a constant in  $\mathbf{V}$ , or an object identifier in  $\text{Oop}$ .

The object wrapper is build by the following function taking a dictionary  $\Delta$  and returning a wrapper:

*make-wrapper* =  
 $\lambda \Delta. \lambda \phi. \lambda \varphi. \lambda m.$

```

case  $tmp = \Delta(m)$  of
isLoc?  $\Rightarrow$ 
  case  $l = tmp|_{Loc}$  of
    isO?  $\Rightarrow in((\mathbf{T} \rightarrow \mathbf{Fun}) + \mathbf{O})\{$ 
       $\lambda\tau.\lambda(\mu, \sigma).\lambda\pi^*.\perp_{(EV \odot Om \odot S)}$  ,
    isPointer?  $\Rightarrow$ 
       $in((\mathbf{T} \rightarrow \mathbf{Fun}) + \mathbf{O})\{$ 
         $\lambda\tau.$  if  $\tau$  then
           $\lambda(\mu, \sigma).\lambda\pi^*.$ 
          case  $\vartheta = \sigma(l)|_{SV}|_{RV}$  of
            isOop?  $\Rightarrow$  case  $\alpha = \mu(\vartheta)|_{Oop}|_{OmV}$  of
              isMeth?  $\Rightarrow \alpha|_{Meth}(\phi, \varphi)(\mu, \sigma)(\pi^*),$ 
              isMObject?  $\Rightarrow (inEV(\vartheta), \mu, \sigma)$ 
            endcase ,
            isV?  $\Rightarrow (inEV(\vartheta), \mu, \sigma)$ 
          endcase
        else  $\lambda(\mu, \sigma)\lambda\pi^*.(unspecified, \mu, store(l, \pi^* \downarrow_1, \sigma))$ 
        endif )
      endcase ,
    isT?  $\Rightarrow in((\mathbf{T} \rightarrow \mathbf{Fun}) + \mathbf{O})(\perp_O)$ 
  endcase

```

Recall that the purpose of a wrapper is to correctly bind the self and the super within what denotes an object in the model, namely a function from selectors into the denotation of slot application, which is itself a function.

The object obtained from the above wrapper will respond to a selector in the following way. If the selector is unknown, bottom is returned (things can go wrong if either there is no slot corresponding to the selector or if, for some reason, the location recorded in the dictionary is invalid). If the selector corresponds to a valid location, the object returns a function  $\mathbf{T} \rightarrow \mathbf{Fun}$ . This function implements mutation of objects: if this function is passed **true**, then a function applying the slot value is returned, but if **false** is passed, it is a function updating the slot value that is returned. In both cases, the returned function  $f$  is in domain **Fun**.  $f$  takes an object memory (**Om**) and a store (**S**) and returns a function taking a list of actuals ( $\mathbf{PV}^*$ ) and returning the result of the message, a new object memory and a new store.

An updating function  $f$  stores the result of the sole actual (extracted from the list  $\pi^*$ ) in the location corresponding to the slot. On the other hand, a non-updating function  $f$  simply returns the value or the object identifier respectively when the slot value is a constant (in **V**) or a memory object (in **MObject**). Methods are denoted by values of domain **Meth**, which are functions taking a *self* and a *super* and returns a function  $f$  in **Fun**. When a slot value is a method, it is applied by passing it the self, the super to get  $f$ , which is executed by passing it the object memory, the store and the actuals.

A method value from **Meth** is created by the function *make-meth* (see Section A.2), which takes a list of formal parameters, a list of expressions (the body of the

method) and an object memory. *Make-meth* allocates the new method in the object memory and returns its corresponding object identifier.

The semantics of a send expression is the following. First, we evaluate the arguments and the receiver of the message. Because the order of evaluation should be irrelevant (unspecified), we follow [3] by applying arbitrary permutations *permute* and *unpermute* (which must be inverses) to the arguments before and after their evaluation. The receiver must evaluate to an object identifier, which is used to get the corresponding memory object in the object memory. If the selector is **'set**, then we have a mutation message, in which case the name of the slot to be mutated is the result of the first actual. We pass this first actual to the object and then we pass **false** to get a function that will mutate the value of the slot in the store. If the selector is not **'set**, then we pass the selector and then **true** to the object to get a function that will either return the slot value or apply the method. The valuation function is given in Figure 5.

The valuation functions for self- and super-expressions are similar, except that instead of accessing the object memory to get the receiver, we use either the argument *self* or *super* as receiver (see Fig. 5).

In the valuation function for send-expressions, the careful reader may have noticed that we first project a memory object on its first component before passing it the selector. In fact, a memory object is a record containing an object (value of domain **Object**) in its first component. It also contains its own generator, the list of its slot names, the list of the corresponding locations in the store and the generator of its parent object. Hence a memory object is a value of the domain:

$$\mathbf{MObject} = \mathbf{Object} \times \mathbf{Generator} \times \mathbf{Sym}^* \\ \times \mathbf{Loc}^* \times \mathbf{Generator}$$

This representation comes from the fact that an object does not only respond to messages. We must be able to create a child object, which means that we need the object generator. We must also be able to clone an object, which means that we can get its slot names, its current slot values (through the locations in the store) and use its parent generator to create the clone. For example, the valuation function for *newInitial*-expressions (see Fig. 5) evaluates the new object's slot values, slot names and parent object and call a function *allocate-new-object* to create the new object. *Allocate-new-object* takes the object identifier of the parent of the new object, a list of slot names, a list of slot values, an object memory and a store. It first allocates the slot values in the store, then it creates the new object and allocates it in the object memory. It finally returns the object identifier, along with the updated object memory and store (see Section A.3).

The function *clone*, called by the valuation function for clone-expressions, also looks up the object memory to retrieve the object to be cloned. It collects the values in the store corresponding to the cloned object and then calls the function *create-object* using the cloned object's slot names, current slot values and parent generator. The new object is then stored in the object memory using a new object identifier (see Section A.3).

## Observations

This completes the semantics of our first language. Other semantic functions are given in Figure 5: they are pretty standard and should be self-describing. A last point worth mentioning concerns the run-time representation of objects in this semantics. The preservation of object generators at run-time is an unsurprising yet important aspect<sup>1</sup>. In practice, it insists on the fact that the pseudo-variable **self** cannot be bound within individual objects, an important difference compared to objects in class-based languages. The late-binding of **self** in class-based languages arises because methods can apply to instances of their defining class or any of its subclasses. Within one specific object, the **self** is perfectly known and an implementation willing to pay the price (in terms of space obviously) may copy into individual objects the methods applying to them. In each copy, all references to **self** can be bound to the object's self. While it is not the standard way to implement references to self, this observation has been used several times to implement run-time optimisations (look at Self's multiple compilation of methods [2]). Prototypes lack this known **self** property.

Even though they are attached to objects, methods in prototype-based languages retain the same kind of late-binding of their references to **self** as in class-based languages, because they can apply to any (new) descendant object. You can copy them to bind **self**, but the object ends up having two copies of the same method, one with an "open **self**" and one with a bound self (this is the Self approach to multiple compilation). However, the object itself has no bound **self**. The **self** is rather bound to the receiver on a per message basis, at message reception time.

## 4 Two kinds of objects, one kind of link

Typically, a language with one kind of object and one kind of link will evolve towards one with two kinds of objects when some objects become exceptional compared to others. Objects can become exceptional because of a particular programming methodology that must be

<sup>1</sup>This has been independently brought to the fore by Steyaert and De Meuter [16] whose discussion is enlightening.

supported at the language level in order to have all its strength. They also become exceptional when their "first-classness" is severely restricted, such as being immutable or abstract (in the sense that they cannot answer messages). They may not be cloned or used as parents. We illustrate this category of languages with a trait-based programming language.

The trait-based programming model [17] proposes to segregate the slots of a prototypical object in two parts: the "internal" representation part and the "external" protocol part. Typically, the external protocol part consists of slots holding method values while the internal part will consist of slots holding either object identifiers or constant values, but this need not to always be the case. In our first example (Figure 1), the internal part of a point object would include the slots **x** and **y**, while the external part would include the slots **display** and **add**. A traits object is an object holding the external part, the idea being that this object can be shared among several copies of the representation part (themselves represented as objects, see Figure 6). Wegner [19, p. 173] had already invented traits defined in the following way:

"The notion of virtual operations of Simula and other class-based languages may be defined more generally for delegation-based languages. Virtual operations arise when an ancestor specifies resources that will be implemented later in a descendant:

**virtual resource (operation):** A resource (operation) named and specified in an ancestor whose implementation will be provided by a descendant."

However, what Wegner as well as the Self team forgot to insist on is that, in prototype-based programming, while assumed to be a concrete object, the traits object will fail if sent the message **add** or **display** since presumably these methods will try to send the receiver (self) messages **x** and **y** which will fail. Wegner noted that: "The dynamic resource sharing provided by delegation has its costs in object autonomy. It is as though objects are connected to ancestors by an umbilical cord which they can never cut." But he should also have added that ancestors themselves are also linked to their children objects by the very same umbilical cord! In fact, for trait-based programming to work properly, traits objects must never receive messages. And we propose that the language must make sure that it happens like that.

So our example of a language with two kinds of objects and one kind of link is a trait-based programming language whose semantics is given below. In this language, traits are abstract objects that cannot receive messages. As an interesting side-effect, preventing traits

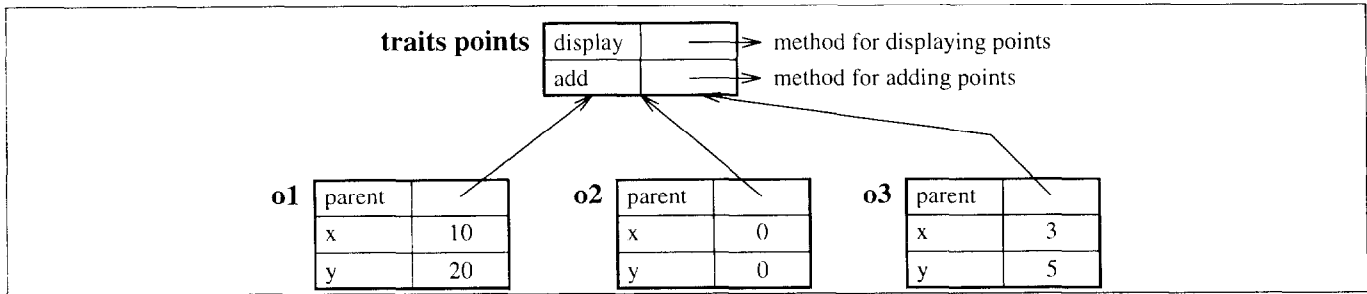


Figure 6: A point example using a traits object.

to answer messages also enforce in some sense the object-centered programming model of the resulting language. Nevertheless, we admit that trait-based programming model is at the extreme abstraction-centered end of the spectrum among the ones presented here.

The trait-based programming methodology encourages the creation of delegation hierarchies which look like an inheritance one in its higher levels made of traits. At the leaf, we find concrete objects, somewhat like the instances in class-based programming. Trait-based programming leaves more flexibility in the creation of objects, but traits are not classes: they are less flexible and abstract. Our trait-based language is minimal; we do not include operations taking advantage of traits. For example, we should add an operation `traitof` returning for an object its first parent traits object. Testing whether two objects support the same protocol would then boil down to test whether or not the two object's traits are the same or have a parent in common. In this sense, trait-based programming encourages a methodology where the notion of similar behavior among objects becomes very important.

### Semantics of trait-based programming

We obtain a trait-based programming language from our previous one by adding one new kind of expressions: the `newTraits` expressions, which create traits objects. The original `newInitials` expressions are preserved to create concrete ones. The essential idea of the semantics is to segregate the domain of memory objects (**MObject**) in two subdomains: a domain of concrete objects (**CObject**) and a domain of traits objects (**TObject**). Hence, we replace the old domain **MObject** by the following:

$$\mathbf{CObject} = \mathbf{Object} \times \mathbf{Generator} \times \mathbf{Sym}^* \times \mathbf{Loc}^* \times \mathbf{Generator}$$

$$\mathbf{TObject} = \mathbf{Generator} \times \mathbf{Sym}^* \times \mathbf{Loc}^* \times \mathbf{Generator}$$

$$\mathbf{MObject} = \mathbf{CObject} \oplus \mathbf{TObject}$$

Notice that because traits will never receive messages, there is no need to keep the object itself (the value of the

domain **Object**) in their representation. In fact, our language imposes two restrictions on the use of traits: traits cannot be sent a message, and they cannot be created as children of a concrete object. The second one is debatable, but this is the way we envisaged it at this time. This leads to a few minor modifications in the previous semantics. Two different functions must be introduced to allocate and create either concrete or traits objects. Cloning must also take into account the status of the cloned object to create a concrete object if the cloned one is concrete or, in the contrary, to create a trait. We also assume the *root* object to be a trait. The section B provides the definitions of the valuation functions for `newInitials`- and `newTraits`-expressions.

The valuation function for `send`-expressions now accepts only concrete objects as receivers:

$$\begin{aligned} \mathcal{E}[\![\text{send } e \ m \ e^*]\!] &= \\ &\lambda(\phi, \varphi). \lambda(\rho, \mu, \sigma). \\ &\text{let}^*(\varepsilon^*, \mu_1, \sigma_1) = \mathcal{RL}[\![\text{permute}(\{e\}e^*)]\!](\phi, \varphi)(\rho, \mu, \sigma) \\ &\text{and } (\varepsilon, \varepsilon_1^*) = \text{split}(\text{unpermute}(\varepsilon^*)) \\ &\text{in if } \mathcal{L}[\![m]\!] \mid_{\text{Sym}=\varepsilon_{\text{Sym}}} \text{set} \\ &\quad \text{then } \text{on}_1(\text{bound-object}(\varepsilon \mid_{\text{RV}|\text{Oop}})(\mu_1) \mid_{\text{MObject}|\text{CObject}}) \\ &\quad \quad (\varepsilon_1^* \downarrow_1 \mid_{\text{RV}|\text{V}|\text{Sym}}) \mid_{\{\text{T} \rightarrow \text{Fun}\}} (\text{false})(\mu_1, \sigma_1)(\varepsilon_1^* \uparrow 1) \\ &\quad \text{else } \text{on}_1(\text{bound-object}(\varepsilon \mid_{\text{RV}|\text{Oop}})(\mu_1) \mid_{\text{MObject}|\text{CObject}}) \\ &\quad \quad (\mathcal{L}[\![m]\!] \mid_{\text{Sym}}) \mid_{\{\text{T} \rightarrow \text{Fun}\}} (\text{true})(\mu_1, \sigma_1)(\varepsilon_1^*) \\ &\quad \text{endif} \end{aligned}$$

Actually, the object that we obtain by looking up the object memory is explicitly projected into **CObject**. This assumes that everything goes right, since we don't deal explicitly with errors in this semantics to keep it simple and more comprehensible.

## 5 Two kinds of objects, two kinds of links

Languages with one kind of link will typically have a delegation, or parent-of link, which implements sharing akin to inheritance. Adding a second kind of link suggests introducing a structural description link similar to the class-of link. In traditional prototype-based languages, each object being one-of-a-kind, it gets both

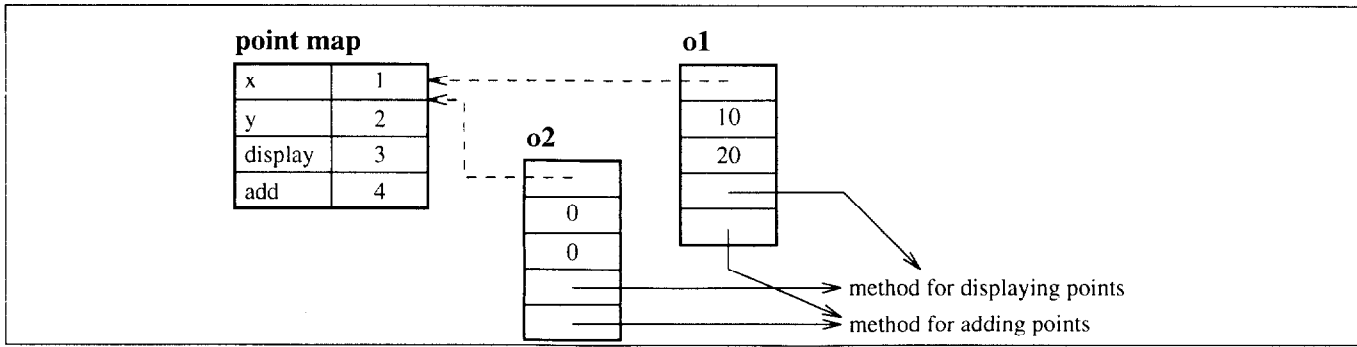


Figure 7: The point example using maps.

its slot names and slot values. When a large number of structurally identical objects are created, it becomes tempting to share the structural information, a line of reasoning that pushed the Self team to invent *maps* [2].

Maps, illustrated in Figure 7, are similar to standard prototypes, except that the slot values are replaced by indexes giving the relative position of the slot values in the object, now represented merely as a vector of slot values. In fact, Self goes beyond that by putting in the map the values of immutable slots, an existing concept in this language. When an object receives a message, the selector is used to look up the map to find a slot index, a value or a method. If a value is found, it is returned; if a method is found, it is applied in the context of the receiver; if an index  $n$  is found, the content of receiver's  $n$ th slot is retrieved and either returned if it is a value or applied if it is a method. In Self, maps are created automatically behind the scene and if an object is cloned, the two clones will share the same map. They give Self the same space efficiency in the representation of objects as the one of class-based languages.

A map-based language is constructed by making maps true objects. Because their slots contain indices to be reinterpreted in the context of the receiver, maps, as objects, cannot answer messages. So we make them abstract objects unable to answer messages. With maps represented as objects, it becomes possible to take advantage of them in the programming methodology. A map-based language encourages a programming methodology where the notion of structurally similar objects becomes very important. In such a language, we can speak about a group of structurally identical objects. Our language is a minimal one, but map-based operations to add or delete a slot to all objects in a clone family, etc. could readily be implemented.

### Semantics of map-based programming

In order for a map-based language to be implemented, we must first have a notion of contiguous locations in the

store as well as a location arithmetic that will allow to implement concrete objects as vectors of slot values. The three functions *locations*, *reservations* and *contiguous-allocations* add to our store algebra the necessary concepts. The function *locations* returns  $n$  contiguous locations, which are used by the functions *reservations* and *contiguous-allocations* (see Section C).

Using this augmented store algebra, we can create a concrete object with  $n$  slots by allocating  $n$  contiguous locations in the store. Also, when cloning an existing object, we can collect the current slot values in  $n$  contiguous locations.

Maps are inferred automatically when an object is created by a `newInitials`-expression. Maps are very similar to objects (values of the domain **Object**) since the implementation will query them with a selector to obtain a value. We therefore implement them using similar techniques, except that there is no need for a wrapper semantics since they don't have to answer real messages and they don't have parents. Also, a map in our language contains only indices giving the position of the corresponding slot value in the concrete object. A map is viewed as a function from selectors into functions of stores into indices:

```

build-map =
  λ(q*, σ).
  let* n* = create-indices(q*)(0)
  and (l*, σ1) = allocate-slot-values(n*)(σ)
  in ( let Δ = make-object-dict(q*)(l*)
      in λm. case tmp = Δ(m) of
          isLoc? ⇒ in((S → Num) + O)(
              λσ2.σ2(tmp|Loc |sv|RV|v|Num) .
          isT? ⇒ in((S → Num) + O)(⊥O)
      endcase .σ1)

```

where the function *create-indices* takes an integer  $n$  and return the list of integers from 1 to  $n$ .

A concrete object on the other hand needs the full wrapper semantics. However, the function *make-wrapper* is slightly different from what we had in the previous

languages because we have to replace the object's dictionary by the map to which they are linked. Hence *make-wrapper* takes the map as first parameter. When the object is activated with a selector, the map is applied to it. If the selector is known, the map returns a function which is applied to the current store to get the corresponding index  $n$ . This index is used to retrieve the  $n$ th slot value in the concrete object. This value is then used as in the previous version of *make-wrapper*:

```

make-wrapper =
  λΠ.λl.λφ.λφ.λm.
  case tmp = Π(m) of
  is(S → Num)? ⇒
    m((T → Fun) + O)(
      λτ. if τ
      then
        λ(μ, σ).λπ*.
          case ϑ = σ(add-to-loc(l,
            tmp |(S→Num)(σ))) |SV|RV of
          isOop? ⇒ case α = μ(ϑ |Oop) |OmV of
            isMeth? ⇒ α |Meth(φ, φ)(μ, σ)(π*) ,
            isMObject? ⇒ (inEV(ϑ), μ, σ)
          endcase ,
          isV? ⇒ (inEV(ϑ), μ, σ)
        endcase
      else
        λ(μ, σ).λπ*.
          (unspecified, μ,
            store(add-to-loc(l, tmp |(S→Num)(σ)), π* ↓1, σ))
        endif ) ,
  isO? ⇒ in((T → Fun) + O)(⊥O)
  endcase

```

The new function *allocate-new-object* first creates a map for the object, allocates the map in the object memory and calls the function *create-object* to create the concrete object. This latter function allocates the slot values, calls *make-wrapper* to create the new wrapper and allocates the concrete object in the object memory:

```

allocate-new-object =
  λ(ε, q*, ε*, μ, σ).
  let* Γ = on2(μ(ε |RV|Oop) |OmV|MObject|CObject)
  and (Π, σ1) = build-map(q*, σ)
  and (ρ, μ1) = allocate-oop(μ)
  and μ2 = bind-object(ρ)(inOmV(inMObject(Π)))(μ1)
  and (mo, σ2) = create-object(Π, ρ, ε*, σ1, Γ)
  and (ρ1, μ3) = allocate-oop(μ2)
  and μ4 = bind-object(ρ1)(inOmV(mo))(μ3)
  in (inEV(inRV(ρ1)), μ4, σ2)

```

```

create-object =
  λ(Π, ρ, ε*, σ, Γ).
  let (l*, σ1) = allocate-n-slot-values(ε*)(σ)
  in (let Γ1 = make-wrapper(Π)(l* ↓1) ▷ Γ
    in inMObject((fix(Γ1), Γ1, ρ, l* ↓1, #ε*, Γ)) , σ1)

```

Memory objects are now represented using the following semantic domains:

$$\begin{aligned} \mathbf{CObject} &= \mathbf{Object} \times \mathbf{Generator} \times \mathbf{Oop} \times \mathbf{Loc} \\ &\quad \times \mathbf{Num} \times \mathbf{Generator} \\ \mathbf{Map} &= \mathbf{Selector} \rightarrow ((\mathbf{S} \rightarrow \mathbf{Num}) + \mathbf{O}) \\ \mathbf{MObject} &= \mathbf{CObject} \oplus \mathbf{Map} \end{aligned}$$

A concrete object contains its object (used to answer messages), its generator (used to create child objects), its map object identifier, its first location, the number of contiguous locations in the object and the parent's generator (used for cloning).

The valuation functions for message passing expressions (send-, self- and super-expressions) are exactly the same as in the trait-based language presented earlier. We simply add two new expressions: a *mapof-expression* (*mapof*  $\epsilon$ ) which retrieves the object identifier of an object's map, and a *clone?-expression* (*clone?*  $\epsilon \ e_1$ ) which tests whether two objects are clones of one another by testing the equality of their respective map's object identifier. The valuation functions are:

$$\begin{aligned} \mathcal{E}[\![\mathbf{mapof} \ \epsilon]\!] &= \\ &\lambda(\phi, \varphi). \lambda(\rho, \mu, \sigma). \\ &\text{let } (\epsilon, \mu_1, \sigma_1) = \mathcal{R}[\![\epsilon]\!](\phi, \varphi)(\rho, \mu, \sigma) \\ &\text{in } (\text{inEV}(\text{inRV}(\text{on}_3(\mu(\epsilon |_{\text{RV}}|_{\text{Oop}}) |_{\text{OmV}}|_{\text{MObject}}|_{\text{CObject}}))), \mu_1, \sigma_1) \\ \mathcal{E}[\![\mathbf{clone?} \ \epsilon \ e_1]\!] &= \\ &\lambda(\phi, \varphi). \lambda(\rho, \mu, \sigma). \\ &\text{let}^* (\epsilon^*, \mu_1, \sigma_1) = \mathcal{R}[\![\text{permute}(\langle \epsilon, e_1 \rangle)]\!](\phi, \varphi)(\rho, \mu, \sigma) \\ &\text{and } \epsilon_1^* = \text{unpermute}(\epsilon^*) \\ &\text{and } \epsilon = \epsilon_1^* \downarrow_1 \\ &\text{and } \epsilon_1 = \epsilon_1^* \uparrow_1 \downarrow_1 \\ &\text{in } (\text{inEV}(\text{inRV}(\text{inV}(\text{on}_3(\mu(\epsilon |_{\text{RV}}|_{\text{Oop}}) |_{\text{OmV}}|_{\text{MObject}}|_{\text{CObject}}) =_{\text{Oop}} \\ &\quad \text{on}_3(\mu(\epsilon_1 |_{\text{RV}}|_{\text{Oop}}) |_{\text{OmV}}|_{\text{MObject}}|_{\text{CObject}}))), \mu_1, \sigma_1) \end{aligned}$$

In the first, we retrieve the object identifier of the map in the object representation. In the second, we test whether two objects are clone by testing if they have the same map. In fact, the property that we test is not structural equivalence but whether or not two objects have been obtained through a series of cloning operations originating on the same object.

## 6 Two kinds of links, one kind of object

Typically, a language with two kinds of links and one kind of object can be obtained by some rationalization of a language with two kinds of objects (and two kinds of links). For example, consider the map-based language proposed earlier. Can we transform maps in order to make them standard objects capable of answering messages without impairing the programming model? The answer is yes. The problem with maps begins when we send them messages whose result needs a reinterpretation in the context of a concrete object in order to

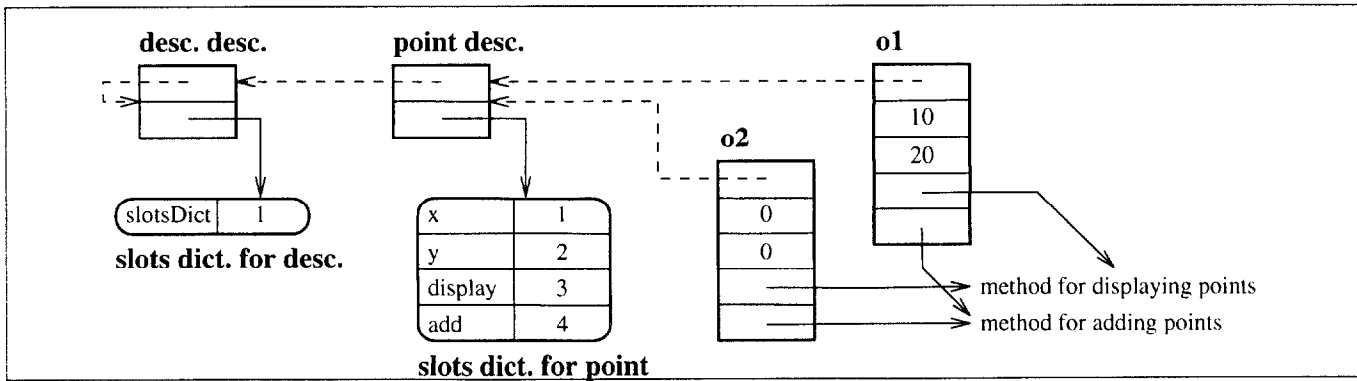


Figure 8: The point example using descriptors.

make sense. But maps don't need to be implemented like standard prototypes. A simple idea, illustrated in Figure 8, is to replace the map by a descriptor object with one slot called `slotsDict`, which points to an object implementing a slots dictionary. The slots dictionary is not an ordinary prototype (despite a similar yet not identical aspect in Figure 8), but rather an object similar to Smalltalk's method dictionary. We draw it as a box with round corners and it is actually an object answering `at:<some selector>` returning its associated value and `at:<some selector> put:<some value>` messages like a Smalltalk dictionary.

The advantage of this representation is that now we can send legitimate messages to descriptors, namely `slotsDict`, as well as to slots dictionary. In fact, our descriptors look pretty much like Smalltalk classes. A descriptor-based language similar to the previous map-based one can be designed very simply. In order to preserve an object-centered programming model, descriptors should be created automatically, like maps were. Nonetheless, descriptors are so much like classes that we are at the frontier between abstraction-centered and object-centered programming here. An important missing feature that makes the language still promote an object-centered programming model is the lack of a sharing mechanism between descriptors that would have a semantics similar to inheritance. By taking care not to introduce such a link between descriptors, we don't encourage programmers to design their applications mainly around descriptors. Due to the lack of space, we don't provide a formal semantics for a descriptor-based programming language (see [11]).

## 7 The classification

The Figure 9 summarizes our classification. We have represented five categories of languages: four described here with one example language in each, and one corre-

sponding to Wegner's object-based languages in which he classifies Ada [6]. The list of possible languages in each class is by no mean closed. For example, another path towards a language with two kinds of objects but one kind of link appears when considering the status of prototypical objects in Lieberman's first proposal. In Lieberman's mind, the standard way to represent a concept is to provide a prototypical instance of this concept (Clyde is the prototypical elephant) to which other objects of the same concept can delegate for default properties. Because modifying a prototypical object has an important, and often undesirable, effect on all other objects delegating to it, we can make them immutable objects, hence stressing their particular role. In the Clyde-Fred example, Clyde would no longer be mutable, therefore making it impossible to indirectly modify Fred by mutating Clyde. A safe version of such a language designed in this line provides another example of a language with two kinds of objects and one kind of link.

It is also worth noting that the classification using the number of kinds of objects and links needs not to be restricted to object-centered languages. We have already noted that our descriptor-based language is at the frontier of abstraction-centered programming. If we take this language and add an inheritance link between descriptors, we end up having a language with one kind of objects but three kinds of links (parent-of, descriptor-of and descriptor inheritance) which cannot be characterized as object-centered. Moreover, consider a language where metaclasses are first-class objects as Cointe's ObjVlisp. Such a language has two kinds of objects (instances and classes, since metaclasses are simply classes whose instances are classes) and two kinds of links: instantiation and inheritance. However, it is certainly not object-centered. We have used our classification to characterize object-centered programming languages, but it is not restricted to this class of languages.

An interesting work now is to assess existing lan-

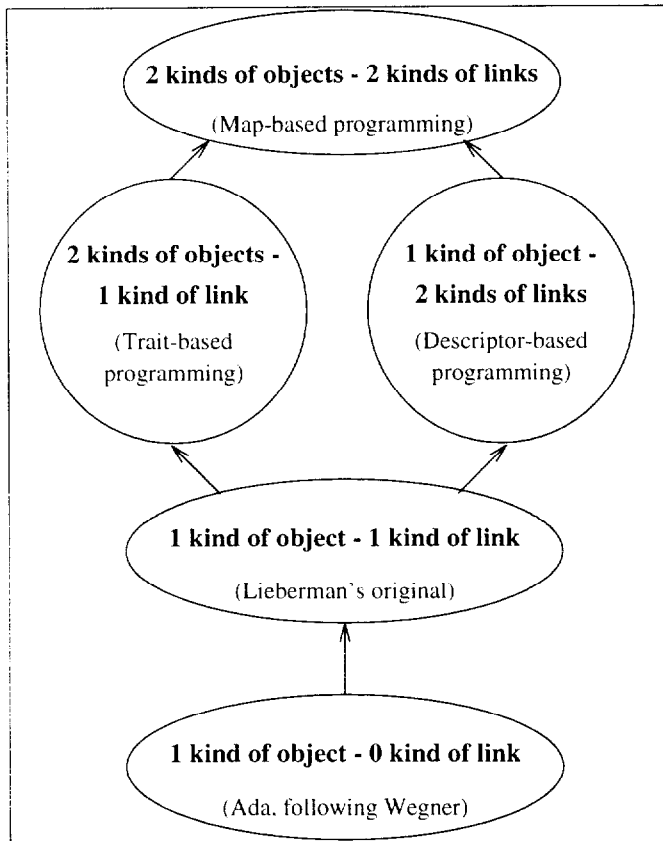


Figure 9: The classification with example languages.

guages in the light of our classification. For example, in our view Self is not prototype-based but it is certainly object-centered. It is our opinion that using the notion of object-centered programming with abstract objects presented here, the design of Self can be positively enhanced. Our preferred path of upgrade would be to introduce traits as abstract objects like in Section 4 but to make maps first-class objects in the line of our descriptors (Section 6). This would make Self a delegation-based programming language with two kinds of objects and two kinds of links.

## 8 Conclusion

In the current classifications of object-oriented programming languages, there is an underlying assumption that delegation-based and prototype-based languages are the same. In fact, this assumption has introduced confusion over the meaning of prototype-based programming. In the first papers proposing the prototype-based approach, authors insist on a programming model admitting only concrete objects. On the other hand, several real prototype-based programming languages have abstract features going against this basic principle.

In this paper, we have contributed towards a better understanding of delegation-based programming languages by introducing the more general notion of object-centered programming, which admits some kinds of abstract devices, such as traits and maps, provided that the programming model is still dominated by the object-centered subset of the language, i.e. the major application design activity revolves around the creation of concrete objects. Using this more general notion, we have proposed a new classification of delegation-based languages, primarily used to show the diversity of programming models we can obtain by introducing different abstract entities into delegation-based programming. Indeed, this classification brings to the fore the existence of more and more structured delegation-based languages forming a continuum between pure prototype-based languages and class-based ones.

Our future work is to assess existing languages in the light of this classification and to study more thoroughly the new programming models introduced by the different languages proposed in this paper.

## References

- [1] BORNING, A. Classes versus Prototypes in Object-Oriented Languages. In *Proc. of the IEEE/ACM Fall Joint Conference* (1986), pp. 36–40.
- [2] CHAMBERS, C., UNGAR, D., AND LEE, E. An Efficient Implementation of Self, a Dynamically-typed Object-Oriented Language Based on Prototypes. *Proc. of OOPSLA '89, Sigplan Not.* 24, 10 (Oct. 1989), 49–70.
- [3] CLINGER, W., AND REES, J., Eds. *Revised<sup>A</sup> Report on the Algorithmic Language Scheme* (Nov. 1991).
- [4] COOK, W., AND PALSBERG, J. A Denotational Semantics of Inheritance and its Correctness. *Proc. of OOPSLA '89, Sigplan Not.* 24, 10 (Oct. 1989), 433–443.
- [5] DONY, C., MALENFANT, J., AND COINTE, P. Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation. *Proc. OOPSLA '92, Sigplan Not.* 27, 10 (Oct. 1992), 201–217.
- [6] ICHBIAH, J. (Ed.) *Ada Programming Language*, ANSI/MIL-STD-1815a. Ada Joint Program Office, Department of Defense, Washington, D.C., 1983.
- [7] HENSE, A. V. Wrapper semantics of an object oriented programming language with state. In *Int. Conf. on Theoretical Aspects of Computer Software, TACS'91* (Sep. 1991), pp. 548–568.
- [8] LALONDE, W. Designing Families of Data Types Using Exanplars. *ACM Trans. on Prog. Languages and Systems* 11, 2 (April 1989), 212–248.
- [9] LIEBERMAN, H. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. *Proc. of OOPSLA '86, Sigplan Not.* 21, 11 (Nov. 1986), 214–223.

- [10] MALENFANT, J. Split objects: Taming value sharing in object-oriented languages. submitted, August 1994.
- [11] MALENFANT, J. Semantics of Delegation-Based Programming Languages. forthcoming tech. report, 1995.
- [12] MOSSES, P. Denotational Semantics. In *Handbook of Theoretical Computer Science*. MIT Press, 1990, ch. 11, pp. 575–631.
- [13] MYERS, B. A., GIUSE, D. A., AND ZANDEN, B. V. Declarative Programming in a Prototype-Instance System: Object-Oriented Programming Without Writing Methods. *Proc. OOPSLA '92, Sigplan Not.* 27, 10 (Oct. 1992), 184–200.
- [14] SCHMIDT, D. *Denotational Semantics: a methodology for language development*. Wm. C. Brown, 1986.
- [15] STEIN, L., LIEBERMAN, H., AND UNGAR, D. A Shared View of Sharing: The Treaty of Orlando. In *Object-Oriented Concepts, Applications and Databases* (1988), W. Kim and F. Lochovsky, Eds., Addison-Wesley.
- [16] STEYAERT, P., AND DE MEUTER, W. A Marriage of Class- and Object-Based Inheritance Without Unwanted Children. In *Proc. of ECOOP'95* (August 1995).
- [17] UNGAR, D., CHAMBERS, C., CHANG, B.-W., AND HÖLZLE, U. Organizing Programs without Classes. *Lisp and Symbolic Computation*, 4 (1991), 223–242.
- [18] UNGAR, D., AND SMITH, R. Self: The Power of Simplicity. *Lisp and Symbolic Computation*, 4 (1991).
- [19] WEGNER, P. Dimensions of Object-Oriented Language Design. *Proc. of OOPSLA '87, Sigplan Not.* 22, 12 (December 1987), 168–182.

## A Additional functions for LIEBERMAN

### A.1 Object dictionaries

*empty-dict* =  $\lambda q.in(\mathbf{Loc} \oplus \mathbf{T})(\mathbf{false})$

*add-slot* =  $\lambda q.\lambda l.\lambda \Delta.\lambda q_1.$  **if**  $q =_{sym} q_1$   
                   **then**  $in(\mathbf{Loc} \oplus \mathbf{T})(l)$   
                   **else**  $\Delta(q_1)$   
                   **endif**

*allocate-slot-values* =  
 $\lambda \varepsilon^*.\lambda \sigma.$  **let**  $(l^*, \sigma_1) = allocations(\#\varepsilon^*)(\sigma)$   
                   **in**  $(l^*, assign(l^*, \varepsilon^*)(\sigma_1))$

*make-object-dict* =  
**fix** $(\lambda f.\lambda q^*.$   
   **fix** $(\lambda g.\lambda l^*.$  **if**  $q^* = \langle \rangle$  **then** *empty-dict*  
                   **else**  $add-slot(q^* \downarrow_1)(l^* \downarrow_1)(f(q^* \uparrow_1)(l^* \uparrow_1))$   
                   **endif**  $)$   
                   **endif**  $)$

### A.2 Method creation

*make-meth* =  
 $\lambda(\nu^*, \varepsilon^*, \mu).$   
**let**  $\gamma = inOmV(\lambda(\phi, \varphi).\lambda(\mu_1, \sigma).\lambda\pi^*.$   
                   **let** $^*(l^*, \sigma_1) = allocations(\#\pi^*)(\sigma)$

**and**  $\sigma_2 = assign(l^*, \pi^*)(\sigma_1)$   
                   **and**  $\rho = bindings(\nu^*, l^*)$   
                   **in**  $\mathcal{R}^*[\varepsilon^*](\phi, \varphi)(\rho, \mu_1, \sigma_2)$   
**in**  $let(\varrho, \mu_1) = allocate-oop(\mu)$   
**in**  $(inEV(inRV(\varrho)).bind-object(\varrho)(\gamma)(\mu_1))$

## Observations

The reader should notice two important points concerning methods. First, methods do not create closures as traditional lambda-expressions. This is our choice because we don't want to close methods over slots local to an object. A method normally accesses only the slot local to the objects in which it is held. If a closure is created at method  $m$ 's creation time, we must be aware that the method expression is executed in the context of another existing object, say  $o1$ . Building a closure would make  $o1$ 's slots visible inside the new method  $m$ , which will be held by another object  $o2$ ; an activation of  $m$  in  $o2$  could then result in side-effects on the slots of  $o1$  and break its encapsulation. Therefore, instead of create a closure for  $m$ , we rely on references to self to access the owner's slots. Second, the representation of methods needs to be passed the self and super in the context of its application. We cannot bind them in the method, since it can apply to all of the latter's descendants. We also insist, following the natural implementation of methods in object-oriented programming, that a method can be shared among several objects.

### A.3 Object creation and cloning

*Allocate-new-object* first looks up the object memory for the parent object and gets its generator. It calls in turn the function *create-object*, which allocates the slot values, builds the new wrapper and returns a memory object. *Allocate-new-object* then obtains from the current object memory a new object identifier for this new memory object, which is finally stored in the object memory:

*allocate-new-object* =  
 $\lambda(\varepsilon, q^*, \varepsilon^*, \mu, \sigma).$   
**let** $^*(o, \Gamma, q^*_1, l^*, \Gamma_1) = \mu(\varepsilon |_{RV} |_{OOP}) |_{OmV} |_{MObject}$   
**and**  $(mo, \sigma_1) = create-object(q^*, \varepsilon^*, \sigma, \Gamma)$   
**and**  $(\varrho, \mu_1) = allocate-oop(\mu)$   
**and**  $\mu_2 = bind-object(\varrho)(inOmV(mo))(\mu_1)$   
**in**  $(inEV(inRV(\varrho)), \mu_2, \sigma_1)$

*create-object* =  
 $\lambda(q^*, \varepsilon^*, \sigma, \Gamma).$   
**let**  $(l^*, \sigma_1) = allocate-slot-values(\varepsilon^*)(\sigma)$   
**in**  $(let \Gamma_1 = make-wrapper(make-object-dict(q^*)(l^*)) \triangleright \Gamma$   
                   **in**  $(fix(\Gamma_1), \Gamma_1, q^*, l^*, \Gamma), \sigma_1)$

*clone* =  
 $\lambda(\varepsilon, \mu, \sigma).$  **let** $^*(o, \Gamma, q^*, l^*, \Gamma_1) = \mu(\varepsilon |_{RV} |_{OOP}) |_{OmV} |_{MObject}$   
**and**  $\omega^* = collect-values(l^*, \sigma)$   
**and**  $(mo, \sigma_1) = create-object(q^*, \omega^*, \sigma, \Gamma_1)$

```

 $\mathcal{L}\mathcal{L} :: \text{Constants}^* \rightarrow \mathbf{V}^*$ 
 $\mathcal{L}\mathcal{L}[k^*] = \text{if } k^* = \langle \rangle \text{ then } \langle \rangle \text{ else } \langle \mathcal{L}[k^* \downarrow_1] \rangle \S \mathcal{L}\mathcal{L}[k^* \uparrow_1] \text{ endif}$ 

 $\mathcal{R} :: \text{Expressions} \rightarrow (\text{Object} \odot \text{Object}) \rightarrow (\text{Env} \odot \text{Om} \odot \mathbf{S}) \rightarrow (\text{EV} \odot \text{Om} \odot \mathbf{S})$ 
 $\mathcal{R}[\epsilon] = \lambda(\phi, \varphi). \lambda(\rho, \mu, \sigma). \text{let } (\varepsilon, \mu_1, \sigma_1) = \mathcal{E}[\epsilon](\phi, \varphi)(\rho, \mu, \sigma)$ 
 $\text{in if isLV}(\varepsilon) \text{ then } (\text{stored}(\varepsilon \mid_{\text{LV}} \mid_{\text{Loc}}, \sigma_1), \mu_1, \sigma_1) \text{ else } (\varepsilon, \mu_1, \sigma_1) \text{ endif}$ 

 $\mathcal{R}^* :: \text{Expressions}^* \rightarrow (\text{Object} \odot \text{Object}) \rightarrow (\text{Env} \odot \text{Om} \odot \mathbf{S}) \rightarrow (\text{EV} \odot \text{Om} \odot \mathbf{S})$ 
 $\mathcal{R}^*[\epsilon^*] = \lambda(\phi, \varphi). \lambda(\rho, \mu, \sigma). \text{if } \# \epsilon^* = 1 \text{ then } \mathcal{R}[\epsilon^* \downarrow_1](\phi, \varphi)(\rho, \mu, \sigma)$ 
 $\text{else let } (\varepsilon, \mu_1, \sigma_1) = \mathcal{R}[\epsilon^* \downarrow_1](\phi, \varphi)(\rho, \mu, \sigma) \text{ in } \mathcal{R}^*[\epsilon^* \uparrow_1](\phi, \varphi)(\rho, \mu_1, \sigma_1) \text{ endif}$ 

 $\mathcal{R}\mathcal{L} :: \text{Expressions}^* \rightarrow (\text{Object} \odot \text{Object}) \rightarrow (\text{Env} \odot \text{Om} \odot \mathbf{S}) \rightarrow (\text{EV}^* \odot \text{Om} \odot \mathbf{S})$ 
 $\mathcal{R}\mathcal{L}[\epsilon^*] = \lambda(\phi, \varphi). \lambda(\rho, \mu, \sigma). \text{if } \# \epsilon^* = 0 \text{ then } (\langle \rangle, \mu, \sigma)$ 
 $\text{else let}^* (\varepsilon, \mu_1, \sigma_1) = \mathcal{R}[\epsilon^* \downarrow_1](\phi, \varphi)(\rho, \mu, \sigma)$ 
 $\text{and } (\varepsilon^*, \mu_2, \sigma_2) = \mathcal{R}\mathcal{L}[\epsilon^* \uparrow_1](\phi, \varphi)(\rho, \mu_1, \sigma_1)$ 
 $\text{in } (\langle \varepsilon \rangle \S \varepsilon^*, \mu_2, \sigma_2) \text{ endif}$ 

 $\mathcal{E}^* :: \text{Expressions}^* \rightarrow (\text{Object} \odot \text{Object}) \rightarrow (\text{Env} \odot \text{Om} \odot \mathbf{S}) \rightarrow (\text{EV} \odot \text{Om} \odot \mathbf{S})$ 
 $\mathcal{E}^*[\epsilon^*] = \lambda(\phi, \varphi). \lambda(\rho, \mu, \sigma). \text{if } \# \epsilon^* = 1 \text{ then } \mathcal{E}[\epsilon^* \downarrow_1](\phi, \varphi)(\rho, \mu, \sigma)$ 
 $\text{else let } (\varepsilon, \mu_1, \sigma_1) = \mathcal{E}[\epsilon^* \downarrow_1](\phi, \varphi)(\rho, \mu, \sigma) \text{ in } \mathcal{E}^*[\epsilon^* \uparrow_1](\phi, \varphi)(\rho, \mu_1, \sigma_1) \text{ endif}$ 

```

Figure 10: Some utilities for the valuation functions in Fig. 5.

```

 $\text{and } (\varrho, \mu_1) = \text{allocate-ooop}(\mu)$ 
 $\text{and } \mu_2 = \text{bind-object}(\varrho)(\text{inOmV}(mo))(\mu_1)$ 
 $\text{in } (\text{inEV}(\text{inRV}(\varrho)), \mu_2, \sigma_1)$ 

 $\text{collect-values} =$ 
 $\lambda(l^*, \sigma). \text{fix}(\lambda f. \lambda l^*_1. \text{if } l^*_1 = \langle \rangle \text{ then } \langle \rangle$ 
 $\text{else } \langle \sigma(l^*_1 \downarrow_1) \mid_{\text{SV}} \rangle \S f(l^*_1 \uparrow_1)$ 
 $\text{endif})(l^*)$ 

```

#### A.4 Initial object root

The first object in the system is the *root* object whose identifier is bound to the variable *root-ooop*. *root-ooop* is also used to bind the root object in the initial object memory, *initial-om*, to be used to evaluate programs:

```

 $\text{root-gen} =$ 
 $\lambda\phi. \lambda m. \text{in}((\mathbf{T} \rightarrow \mathbf{Fun}) + \mathbf{O})(\lambda\tau. \lambda(\mu, \sigma). \lambda\pi^*. \perp_{(\text{EV} \odot \text{Om} \odot \mathbf{S})})$ 
 $\text{root-ooop} = \text{let } (\varrho, \mu) = \text{allocate-ooop}(\text{empty-om}) \text{ in } \varrho$ 
 $\text{initial-om} =$ 
 $\text{let } (\varrho, \mu) = \text{allocate-ooop}(\text{empty-om})$ 
 $\text{in } \text{bind-object}(\varrho)$ 
 $(\text{inOmV}((\text{fix}(\text{root-gen}), \text{root-gen}, \langle \rangle, \langle \rangle, \perp_{\text{Generator}})))$ 
 $(\mu)$ 

```

## B Trait-based programming

*allocate-new-conc-object* and *allocate-new-traits-object* are respectively called by the valuation functions for *newInitials* and *newTraits* expressions:

```

 $\text{parent-gen} = \lambda\alpha. \text{case } mo = \alpha \mid_{\text{MObject}} \text{ of}$ 
 $\text{isCObject?} \Rightarrow \text{on}_2(mo \mid_{\text{CObject}}),$ 
 $\text{isTObject?} \Rightarrow \text{on}_1(mo \mid_{\text{TObject}})$ 
 $\text{endcase}$ 

```

```

 $\text{allocate-new-conc-object} =$ 
 $\lambda(\varepsilon, q^*, \varepsilon^*, \mu, \sigma).$ 
 $\text{let}^* \Gamma = \text{parent-gen}(\mu(\varepsilon \mid_{\text{RV}} \mid_{\text{Oop}}) \mid_{\text{OmV}})$ 
 $\text{and } (mo, \sigma_1) = \text{create-conc-object}(q^*, \varepsilon^*, \sigma, \Gamma)$ 
 $\text{and } (\varrho, \mu_1) = \text{allocate-ooop}(\mu)$ 
 $\text{and } \mu_2 = \text{bind-object}(\varrho)(\text{inOmV}(mo))(\mu_1)$ 
 $\text{in } (\text{inEV}(\text{inRV}(\varrho)), \mu_2, \sigma_1)$ 

```

```

 $\text{allocate-new-traits-object} =$ 
 $\lambda(\varepsilon, q^*, \varepsilon^*, \mu, \sigma).$ 
 $\text{let}^* \Gamma = \text{on}_1(\mu(\varepsilon \mid_{\text{RV}} \mid_{\text{Oop}}) \mid_{\text{OmV}} \mid_{\text{MObject}} \mid_{\text{TObject}})$ 
 $\text{and } (mo, \sigma_1) = \text{create-traits-object}(q^*, \varepsilon^*, \sigma, \Gamma)$ 
 $\text{and } (\varrho, \mu_1) = \text{allocate-ooop}(\mu)$ 
 $\text{and } \mu_2 = \text{bind-object}(\varrho)(\text{inOmV}(mo))(\mu_1)$ 
 $\text{in } (\text{inEV}(\text{inRV}(\varrho)), \mu_2, \sigma_1)$ 

```

We also have to introduce two functions for the creation of new objects, one for concrete objects, the other for traits:

```

 $\text{create-conc-object} =$ 
 $\lambda(q^*, \varepsilon^*, \sigma, \Gamma).$ 
 $\text{let } (l^*, \sigma_1) = \text{allocate-slot-values}(\varepsilon^*)(\sigma)$ 
 $\text{in } (\text{let } \Gamma_1 = \text{make-wrapper}(\text{make-object-dict}(q^*)(l^*)) \triangleright \Gamma$ 
 $\text{in } \text{inMObject}((\text{fix}(\Gamma_1), \Gamma_1, q^*, l^*, \Gamma)) , \sigma_1)$ 

```

```

 $\text{create-traits-object} =$ 
 $\lambda(q^*, \varepsilon^*, \sigma, \Gamma).$ 
 $\text{let } (l^*, \sigma_1) = \text{allocate-slot-values}(\varepsilon^*)(\sigma)$ 
 $\text{in } (\text{let } \Gamma_1 = \text{make-wrapper}(\text{make-object-dict}(q^*)(l^*)) \triangleright \Gamma$ 
 $\text{in } \text{inMObject}((\Gamma_1, q^*, l^*, \Gamma)) , \sigma_1)$ 

```

The function *clone* now creates a concrete object when a concrete one is cloned, and a traits if it's a traits that is cloned. Also, the first object root will now be an empty traits object:

```

clone =
  λ(ε, μ, σ).
  let tmp = μ(ε |RV|oop)
  in case mo = tmp |omv|MObject of
    isCObject? ⇒
      let* (o, Γ, q*, l*, Γ1) = mo |CObject
      and ω* = collect-values(l*, σ)
      and (mo1, σ1) = create-conc-object(q*, ω*, σ, Γ1)
      and (ρ, μ1) = allocate-oop(μ)
      and μ2 = bind-object(ρ)(inOmV(mo1))(μ1)
      in (inEV(inRV(ρ)), μ2, σ1) ,
    isTObject? ⇒
      let* (Γ, q*, l*, Γ1) = mo |TObject
      and ω* = collect-values(l*, σ)
      and (mo1, σ1) = create-traits-object(q*, ω*, σ, Γ1)
      and (ρ, μ1) = allocate-oop(μ)
      and μ2 = bind-object(ρ)(inOmV(mo1))(μ1)
      in (inEV(inRV(ρ)), μ2, σ1)
  endcase

```

```

initial-om =
  let (ρ, μ) = allocate-oop(empty-om)
  in bind-object(ρ)
    (inOmV(inMObject((root-gen, ⟨⟩, ⟨⟩, ⊥Generator))))
    (μ)

```

The valuation functions for newInitials- and newTraits-expressions are:

```

ℰ[[newInitials e (m*) (e*)]] =
  λ(φ, φ).λ(ρ, μ, σ).
  let* (ε*, μ1, σ1) = ℛℒ[[permute(⟨e⟩§e*)]](φ, φ)(ρ, μ, σ)
  and (ε, ε*) = split(unpermute(ε*))
  in allocate-new-conc-object(ε, ℒℒ[[m*]], ε*, μ1, σ1)

```

```

ℰ[[newTraits e (m*) (e*)]] =
  λ(φ, φ).λ(ρ, μ, σ).
  let* (ε*, μ1, σ1) = ℛℒ[[permute(⟨e⟩§e*)]](φ, φ)(ρ, μ, σ)
  and (ε, ε*) = split(unpermute(ε*))
  in allocate-new-traits-object(ε, ℒℒ[[m*]], ε*, μ1, σ1)

```

## C Map-based programming

The three following functions add to our store algebra the concepts of contiguous locations:

```

locations(n, σ) = unspecified
reservations =
  . fix(λf.λl*.fix(λg.λσ. if l* = ⟨⟩ then σ
    else f(l* † 1)(reservation(l* † 1, σ))
    endif ))
contiguous-allocations =
  λn. if n = 0 then λσ.⟨⟩, σ
  else λσ. let* l* = locations(n, σ)
    and σ1 = reservations(l*)(σ)
    in (l*, σ1)
endif

```

*Allocate-n-slot-values* allocates  $n$  slot values in the store while *collect-n-values* collects the values of  $n$  contiguous locations:

```

allocate-n-slot-values =
  λε*.λσ. let (l*, σ1) = contiguous-allocations(#ε*)(σ)
  in (l*, assign(l*, ε*)(σ1))
collect-n-values =
  λ(l, n, σ).fix(λf.λn1. if n = n1 then ⟨⟩
    else ⟨σ(add-to-loc(l, n1)) |sv⟩§f(n1 + 1)
    endif )(0)

```

The function *create-indices* takes an integer  $n$  and return the list of integers from  $1$  to  $n$ :

```

create-indices =
  fix(λf.λq*.fix(λg.λn. if #q* = 0 then ⟨⟩
    else ⟨n⟩§f(q* † 1)(n + 1)
    endif ))

```

Cloning is defined by the following function:

```

clone =
  λ(ε, μ, σ).
  let* (o, Γ, ρ, l, n, Γ1) = μ(ε |RV|oop) |omv|MObject|CObject
  and ω* = collect-n-values(l, n, σ)
  and (mo, σ1) = create-object(μ(ρ) |omv|MObject|Map,
    ρ, ω*, σ, Γ1)
  and (ρ1, μ1) = allocate-oop(μ)
  and μ2 = bind-object(ρ1)(inOmV(mo))(μ1)
  in (inEV(inRV(ρ1)), μ2, σ1)

```

Finally, *root* must now be a concrete object:

```

initial-om =
  let (ρ, μ) = allocate-oop(empty-om)
  in bind-object(ρ)
    (inOmV(inMObject((fix(root-gen), root-gen, ⊥oop,
      ⊥loc, 0, ⊥Generator))))(μ)

```

## D Object memory functions adapted from the store algebra of [12]

```

empty-om = λρ.in(OmV ⊕ T)(false)
reserve-oop =
  λρ.λμ.λρ1.
  if ρ = oop ρ1 then in(OmV ⊕ T)(true) else μ(ρ1) endif
bind-object =
  λρ.λα.λμ.λρ1.
  if ρ = oop ρ1 then in(OmV ⊕ T)(α) else μ(ρ1) endif
bound-object = λρ.λμ. case tmp = μ(ρ) of
  isOmV? ⇒ tmp |omv,
  isT? ⇒ ⊥omv
  endcase
new-oop(μ) = unspecified
allocate-oop = λμ. let ρ = new-oop(μ)
  in (ρ, reserve-oop(ρ)(μ))

```