Prototalk : A framework for the design and operational evaluation of prototype-based languages

Christophe Dony,
LIRMM – Montpellier-II University

<Abstract

The paper presents a Smalltalk-80 framework dedicated to an operational simulation and evaluation of prototype-based languages. Its principle could be applied to the comparison of any set of similar languages. The ability to reuse code is widely used and the framework architecture makes it possible to implement very easily an object model and an evaluator for a new language by deriving and extending classes representing other similar languages.
Prototype-based languages are currently proposed as a substitute to class-based languages for a higher flexibility in manipulating objects. These languages all revolve around the same basic ideas : object-centered representation, dynamic addition of behavior and slots, cloning operations and a message delegation mechanism. As for any emerging technology, existing languages propose variations on the main concepts. Our framework includes a taxonomy of interpreters for the widely known prototype-based languages. Basic classes implement some "minimal" kernel languages and alternative models such as Self, Act1, ObjectLisp and Actra's examplars are derived from them

1. Introduction

The prototalk platform described in this paper is a Smalltalk-80 framework the general goal of which is to explore the semantics of alternative implementations of the main concepts involved in prototype-based programming. This is a part of a research effort to understand the primitive concepts involved in the design of prototype-based languages, to explore alternatives in their implementation, and to classify prototype-based programming languages on the basis of their respective properties. The analysis made with this platform is presented in [DMC92] but the description of the framework itself has never been published. The platform's goal is neither to fully implement all prototype-based languages, nor to provide efficient implementations of them, but rather to experiment, to focus on determining aspects and to forget incidental ones.

The platform fully exploits the reuse capabilities offered by object-oriented inclusion polymorphism. It is a framework because it offers an architecture to easily implement new languages. Its principle could be applied to the implementation of any kind of languages.

A language is globally represented in the platform by three classes: a first one defining and implementing its object model, a second one implementing an evaluator and a third one allowing to create a workspace in which a toplevel allows expressions of the langage to be entered and evaluated. To implement a new langage consists in

subclassing those classes that represents the closest language already implemented.

The next section briefly recall what is proototype-based programming. Section 4 describes the implementation and use of one basic prototype-bases language. Section ICI presents the global architecture of the framework. Section ICI shows how a new langage can be implemented using the framework.

2. Prototype-based programming

Prototype-based languages propose a new programming paradigm that is justified in two fundamental ways compared to more classical class-based languages. First, on the philosophical side, people's natural way to grasp new concepts is generally to begin by creating concrete examples of these concepts rather than abstract descriptions; class-based languages force to work in the opposite direction by creating abstractions (classes) prior concrete objects (instances). Second, on the more pragmatical side, class-based languages seem to unnecessarily constrain objects, by disallowing to have distinctive behavior for individual objects among instances of a class, and by forbidding inheritance between objects, to share values of instance variables.

Two fundamental concepts found the prototype approach: **autonomous representation** of individual objects and **sharing** (or reuse) between similar individuals based on prototypical examples of concepts. At first, a prototype seems to be some elected object, yet any object can be a prototype [KhAb90, p. 128], and be considered as an example of its own particular concept. Also, autonomy of objects is very important. Prototypes are not meant to be descriptions of concepts, as classes are, and they are not linked in any way to another object that would describe them, as in the class-based approach. Prototypes are concrete examples that represent concepts, and any reference to a concept must first be rephrased in terms of its concrete example, as pointed out by Lieberman [Lieb86]:

> "If Clyde was the elephant most familiar to you, the prototypical elephant might be an image of Clyde himself. If I ask a question such as "How many legs does an elephant have?", a way to answer the question is to assume that the answer is the same as how many legs Clyde has, unless there is a good reason to think otherwise."

1.2. Motivation

Compared to the class-instance approach, prototype-based languages seems to provide a simpler and more flexible programming model, and this flexibility is actually appreciated in applications like user interfaces [MGDV90] as well as virtual reality systems [Born81, Smit86]. Many languages using prototypes have been proposed since a few years. Borning derived a small prototype-based language to compare them to classes [Born86]. Lieberman [Lieb86] gave an informal description of its Delegation system, from which he argued that prototypes are strictly more powerful than classes. Self [UnSm87] is a pure prototype-based language efficiently

implemented [ChUL89, HoCU91, ChUn91] and openly distributed [HCCU90]. Systems mixing prototypes and classes have also been proposed: Lalonde's Examplars [LaTP86, Lalo89] and Stein's Hybrid language [Stei87, Stei89].

However, the simplicity of prototypes is only apparent. Behind a common terminology, many different interpretations of the primitive concepts exist. The current languages all revolve around the same few implementation mechanisms: objects blending behavior and state, dynamic addition of new behavior or state to objects, cloning of existing objects, and delegation. But, concrete implementations of these mechanisms differ in many ways, and this situation makes the understanding of the prototype-based approach quite difficult. Furthermore, current prototype-based languages are either described in an informal way [Lieb86], leaving many crucial implementation issues unresolved[1], or tied to their operational semantics in such a way that one can hardly understand their behavior without having an intimate understanding of all aspects of their implementation[2] [UnSm87].

The large spectrum of variations in the definition of its fundamental concepts plays a large role in the problem we face when trying to understand the prototypebased approach. Cloning, for example, can be interpreted as shallow copy, deep copy, or a mix of both; it is quite clear that the resulting objects will not have the same properties. Delegation, as sharing mechanism, also has many alternatives for its implementation. How do object identify the prototypes they will delegate to? Should delegation be implicit (done automatically by the evaluator) or explicit (done by the object)? Answers to these questions can have important effects on the properties of resulting programming model. Its is our conviction that to achieve the goals of getting a full understanding of prototypes, to compare and assess alternatives, we must also consider the implementation mechanisms, since they have a profound impact on the properties of the resulting model. This is the main motivation of this paper.

1.3. A Smalltalk-80 framework for rapid implementation of prototype-based languages.

Thus, facing such fragmented and divergent approaches, a clear definition of the prototype-based model is definitely needed. The Treaty of Orlando [StLU88] has given a classification of class-based and prototype-based models. However, the Treaty of Orlando do not consider the influence of many crucial implementation choices. In this paper, we propose another way to clarify semantic issues and to obtain a good understanding of these multiple approaches: to implement them within

[1] such as the treatment of the pseudo-variable self in delegation, the choice between implicit and explicit delegation, as well as its implementation.

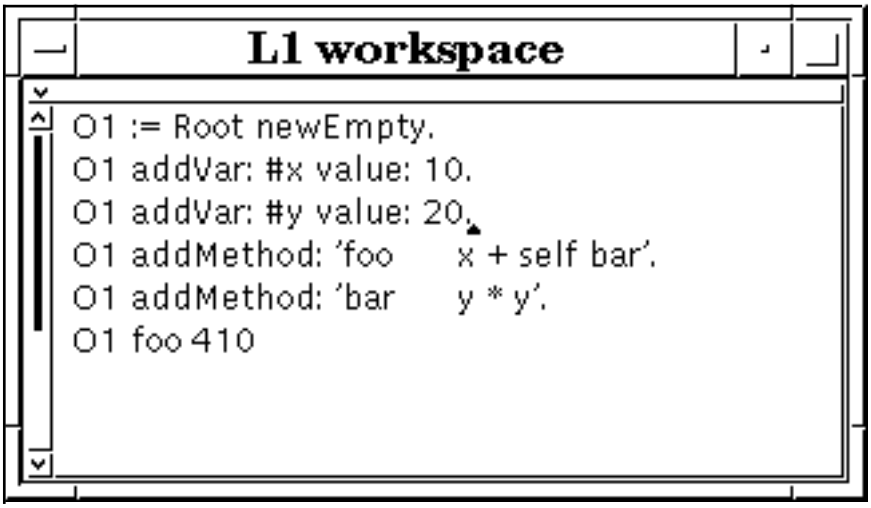
[2] In Self, its is almost impossible to predict the behavior of an object without knowing the entire subgraph of its parent objects as well as the exact implementation of the lookup algorithm, in particular if features like multiple parents and dynamic inheritance are used [ICI].

a common framework, in order to compare, and to show relationships between them. Therefore, we have implemented a Smalltalk-80[3] platform to achieve this goal.

Exemple of a language

In order to make the framework's architecture understandable, this section superficially describes the implementation of one very simple language; since it does not correspond to any known langugage, we simply named it L1. With such a language, users can define new empty objects or clone existing ones, add them variables and methods, send them messages to execute either user-defined methods or langage primitives. Variables can be accessed inside methods. All Smalltalk litteral object (numbers, characters, etc) as well as a predefined prototype, named Root, are available. The syntaxic constructs are variable reference, assignment and message sending. The syntax is Smalltalk's one.

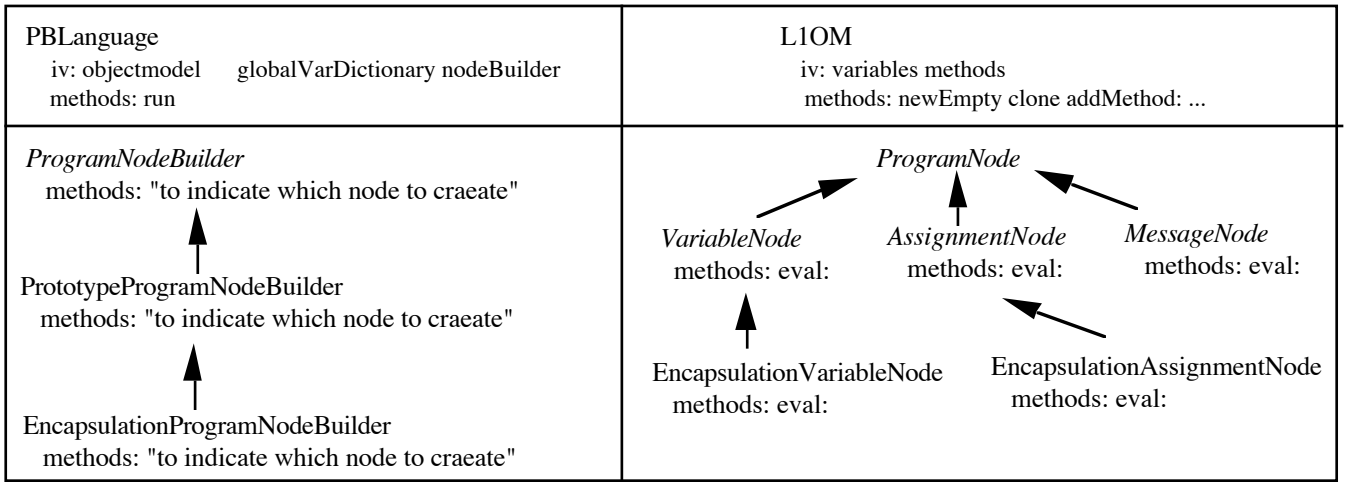
Although very poor, this language is usable; the following figure ICI shows an example of a session. The figure ICI shows a snapshot of a L1 workspace in our platform. The prototype *O1* is created by sending the message *newEmpty* to *Root* . The *Root* prototype is created at bootstrap in order to allow users to start working. Two methods: *x*, *y*, *foo* and *bar* are defined on *O1*. The pseudo-variable *self* usually represents in each method the current receiver.



Overview of a language implementation

The set of classes involved in the implementation of L1 are summarized in figure ICI.

[3]Smalltalk-80 is prefered to existing prototype-based languages, because none of the latters provide a similar programming environment.



First, each langage is represented by an instance of a class named *PBLanguage* (which stands for "prototype-based language") which owns the following instance variables: *objectModel*, *globalVarDictionary* and *nodeBuilder*.

- *objectModel* is a pointer to the class defining the object model.
- *GlobalVarDictionary* is a dictionarry containing global variables used within a sessionwith the language. When a session begins, this dictionary contains at least one variable named *Root* referencing a predefined object, which is an instance of the class stored in the variable *objectModel*. This dictionary can be compared to the Smalltalk-80 global dictionaty containg associations between classes names and classes objects and other global entities. Concerning prototype-based languages, there are no classes and the first objects will be created by the programmer by cloning or extending the *Root* object.
- *nodeBuilder* is a kind of*ProgramNodeBuilder* used to parse user-defined methods of the prototype-based languages.

Object Model

The object model not only determines the structure of prototypes but also the set of primitives available to users to manipulate them. For examples, prototypes in L1 have variables and methods; empty or initialized prototypes can be created with the primitives *newEmpty* and *newInitial:*; variable and methods can be added (primitive *addMethod:* and *addVariable:*). The object model of the language L1 is defined by the class *L1ObjectModel*.which defines two instance variables named *variables* and *methods*. A prototype in L1 is implemented as an instance of *L1ObjectModel*. Its two instance variables (as a Smalltalk object) are initalized with Smalltalk dictionaries to hold its variables and methods (as a prototype). The class *L1ObjectModel* also defines two sets of methods.

- The first one is the set of L1 language primitives, for example *newEmpty, clone*, add*method:* or *addVar:value:*. These primitives, when encountered in a L1

program will of course be computed by the Smalltalk interpreter[4]. To make these primitives available to users, they are inserted in the method dictionary of the Root object of each language.

- The second one is the set of methods used in the evaluation process such as *varValue:* which returns the value of a variable of a prototype.

2.1.1 Internal representation of user-defined methods.

The L1 methods *foo* and *bar* (see figure ICI) internal representation are syntax trees generated by the Smalltalk-80 Parser[5]. Such trees are first-class entities; their nodes being instances of subclasses of the class *ProgramNode*. For example, an instance of the class *MessageNode* is created when the parser finds a message sending instruction. Furthermore, the evaluation of variables read and write access in L1 is different from the standard, thus specific nodes classes (EncapsulationVariableNode and EncapsulationAssignmentNodes) have been created with specific evaluation methods. Which kind of nodes are generated for each kind of instruction of a language is determined by an object, a kind of *ProgramNodeBuilder*, passed as an an argument to the parser. To each class representing a language in our platform is thus associated a program node builder. To L1 is associated an instance of EncapsulationProgramNodeBuilder, a subclass[6] of *PrototypeProgramNodeBuilder*. The figure 2 shows the internal representation of the above method *foo*.

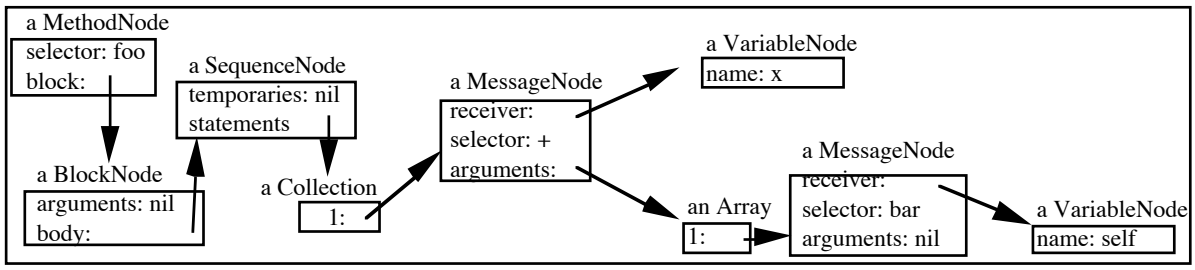


Figure 2: internal representation of the method: bar

2.1.2 An insights into the evaluator.

Each evaluator in the platform is a set of *eval:* methods, defined on subclasses of *programNode*[7], each one being tailored to the interpretation of a particular syntactic construct of the language. An evaluator is thus splitted as in [Lieb87] in several methods. Advantages of this object-oriented representation are (1) that different evaluators for different languages can share some *eval:* methods, for example the

---

[4]This makes no problem since there is a one to one correspondence between a prototype and the Smalltalk object that implements it.

[5]To be able to reuse the Smalltalk parser, we chose to use a Smalltalk-like syntax for all our prototype-based languages.

[6]As far as the syntaxic constructts of VM-PBL are similar to Smalltalk's one, this first subclass of ProgramNodeBuiulder does not redefine any method but is designed as an autonomous root of the subhierarchy of future program node builders dedicated to other prototype-based languages.

[7]Except for the classes *MethodNode* and *BlockNode* which have an *apply* method instead.

eval: method on the class *literalNode* will be shared by all evaluators, and (2) that it is possible to specialize each part of an evaluator independently of the others. Aside from this, the evaluation process is very classical; as an example, let us simply describe the *eval:* message sending method, used for L1 and defined on the class *MessageNode,* in order to compare it with a further version that will implement implicit delegation.

```
eval[8]: context     "defined on the class MessageNode"
    | method rec args newContext |                "temporaries"
1   rec := receiver eval: context.                "receiver is evaluated"
2   method := rec methodNamed: selector.                    "searches the method in the receiver"
3   if   method equals nil
4        then the exception doesNotUnderstand is raised
5        else     args := arguments evlis: context.        "the arguments are evaluated"
6                 if   the method is a smalltalk compiled method
7                      then    this is a message to a litteral or a primitive[9]
8                              return the result of applying the smalltalk method to its arguments
9                      else                                "this is a user-defined method"
10                         newContext := PContext new.     "I create a new context"
11                         newContext at: #self put: rec.  "In which self is the new receiver"
12                         return (method applyWith: args in: newContext)

methodNamed: name    "defined on the class L1"
    I return: (methods at: name ifAbsent: [nil])
```

Figure 3: evaluation of message sending for the L1 language.

The syntax is a mix of Smalltalk's messages sending, assignmentsand pseudo-code.To evaluate a message sending instruction in basic-proto (fig. 3) amounts to search a method in the receiver and to apply it. When the method to be applied has been found, a new context is created in which "self" is bound to the new receiver. The method methodNamed:, defined on L1, is responsible for finding methods in objects.

Interface

Framework architecture

The framework is organized around three main classes hierarchies: the hierarchy of object models classes, the hierarchy of program node builders classes and the hierarchy of program nodes classes holding the evaluators. To implement an interpreter for a new language amount creates an instance of *PBLanguage* and to initialize its instance variables with an instance of an object model class and an instance of a program node builder class. It is possible to choose either existing classes or to insert at the right places new classes in those three hierarchies. As usual with frameworks, the hardest task is certainly to choose this right place, which requires a good understanding of the whole platform.

Different existing (self, ObjectLisp, Exemplars) or "imaginary" languages are already simulates in the platform. We now describe the hierarchy architecture and

[8]All methods *eval:* have an argument which holds the lexical context in which the related instruction has to be evaluated. This lexical context is a dictionary including the arguments, the temporaries and the current value of "self".

[9]In order to make our platform usable rapidly, we have included the possibility to send Smalltalk messages in our methods. For example in the method *bar*, the message + is send to the result of "self x".

give examples of how a new langage can be implemented.

Basic hierarchies

The above part of hierarchies is of great importance, the top classes have to be as general (abstract) as possible in order to allow any level of specialisation.

Object models hierarchy.

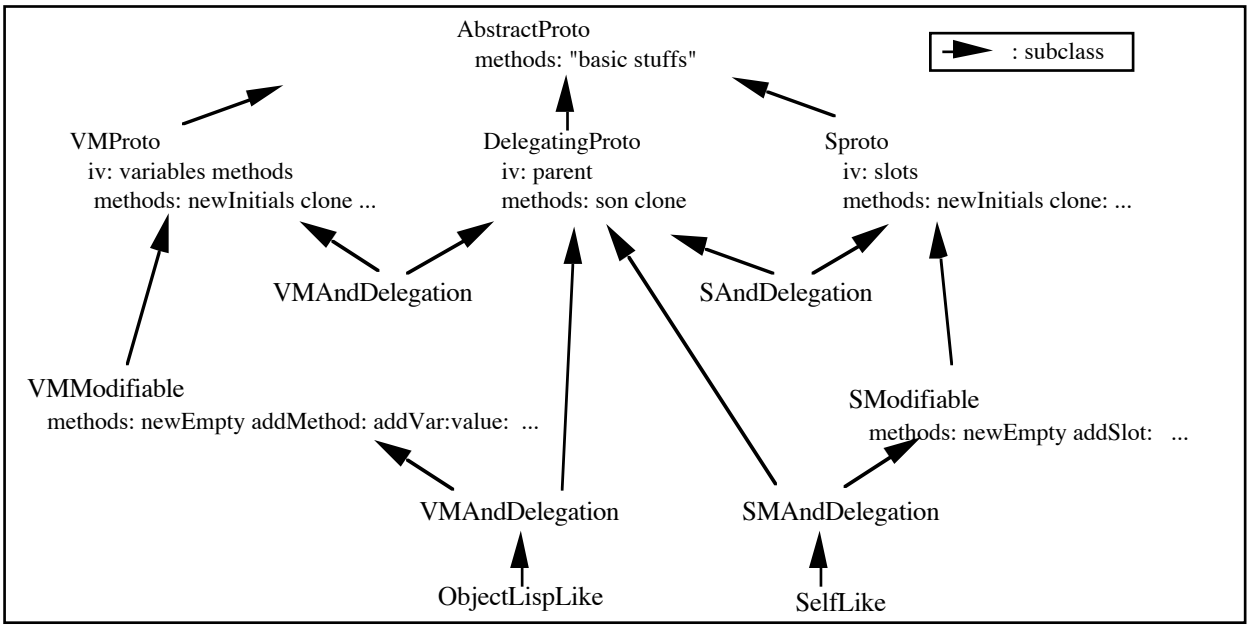


Fig. x12: A subpart of object models hierarchy

A subpart of the hierarchy is shown in figure x12. The top of the hierarchy is made of a set of abstract or very specific classes implementing various basic options.The hierarchy's root class (*AbstractProto*) is an abstract class that do not represent a concrete strucure for prototypes but defines and factorizes method allowing to manipulate prototypes independantly of their structure. Of course most of these methods use other deffered (or "subclass-responsibility" ones.VMProto represents prototypes in languages that handles variables and methods differently as ObjectLisp [ICI] or Exemplar; for prototypes corresponding to that model, variable and methods can only be defined at prototypes creation time (method *newInitials:*). Sproto is for those prototypes that unify variables and methods under the notion of slots as in Self, at the implementation level in our platform prototypes only have a slot dictionary and creation primitive newInitials does a different job. DelegatingProto adds a parent to the prototypes to allow differential creation (method *son*) and delegation.

Program node builders and program nodes hierarchies

Each new evaluator is inheriting its implementation from a previous one, except for those constructions that are given a different semantics in the new language, thus needing their *eval* method to be redefined. The hard task when implementing a new evaluator is therefore to find where it should be inserted in the hierarchy.

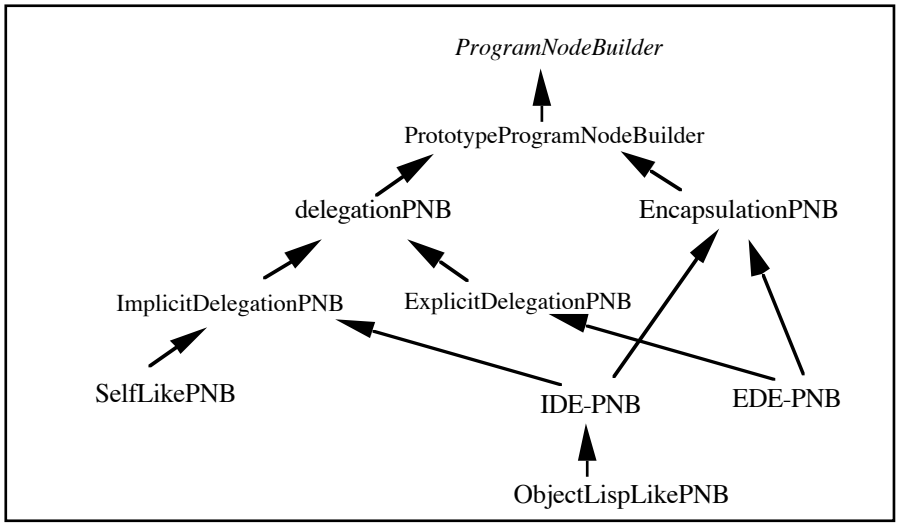


Figure x13: a subpart of the programNodeBuilder hierarchy

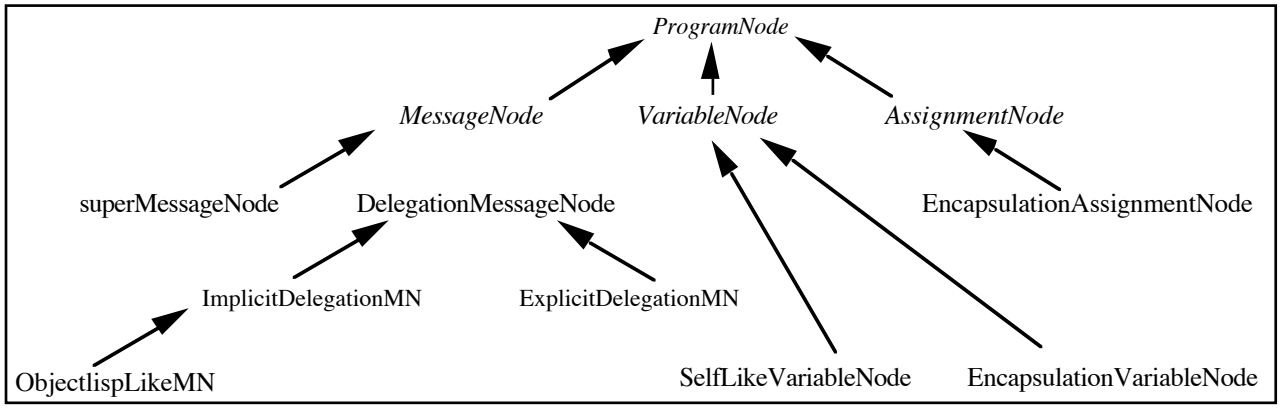


Figure x14: a subpart of the program nodes hierarchy

From an implementation point of view, cloning and delegation have distinct natures. Cloning is performed via message sending and simply requires the definition of a new primitive whereas delegation cannot be implemented with message sending [Lieb86] and supposes modifications in the evaluator . Delegation requires that within the applied method, the pseudo-variable “self” be bound to the client (i.e. the initial receiver of the message) rather than to the object in which the method was found.

Two forms of delegation have been identified [StLU 87]: implicit and explicit. With implicit delegation, the system is responsible for pursuing the search in the shared part of the object, identified by the parent link, when a slot has not been found in the personal part. With Explicit delegation, no internal mechanism is provided to achieve delegation automatically: when a client does not hold a method, it is responsible for saying if it wants to delegate the message and, if so, for explicitly pointing out the server for this message. Implicit and explicit delegation share the above characteristics but vary in their implementations.

3. Differential creation, sharing and reutilisation

Just creating objects from scratch is far from the expressive power expected of an object-oriented language. The questions that arise now are: how to built new objects out of existing ones? How to share knowledge common to different objects? And how to make existing objects reusable (i.e. sharable with potential future objects). For all of them, the notion of differentiation is fundamental. Cloning and delegation have been proposed in prototype-based languages as fundamental mechanisms to create new objects from others either by copying and modifying or by expressing differences.

Cloning (see the primitive "create" in Act1 or "clone" is Self) is a copy operation that avoids the creation of new objects from scratch provided that objects of the same kind already exist in the system. Cloning makes it possible to reuse the design of the structure and the implementation of methods of the cloned object (let us call it the model). Cloning duplicates the structure of the model but ensure slots values sharing at creation time.

Cloning becomes slightly more complicated when split objects and delegation are introduced in the language.

3.2. Delegation

Delegation has been introduced [Lieb81] as a mechanism to retrieve and reuse knowledge shared by different objects. Delegation thus supposes the ability to define objects having a shared part and a personal part:

> "To create an object that shares knowledge with a prototype, you construct an extension object, which has a list containing its prototypes, which may be shared by other objects, and personal behaviour idiosyncratic to the object itself." [Lieb86]

The shared part can be any other object in the system and the personal part defines the slots of the new object that are not in the shared part or that differ from those stored in the shared part. Prototypes can thus be defined by similarity with, or distinction from, other prototypes. For example, in Act1, the primitive "extend" creates a new object, the shared part of which being the receiver of the extend message. The personal part knows about its shared part through a link named "proxy". In Self, the creation of an empty object to which is added a slot named "parent" assigned to an existing object is conceptually equivalent to "extend". In both cases, the parent or proxy link is used to retrieve the shared part. The most important point here is that sharing is done at the level of concrete objects and not at the level of concepts as with class-inheritance; this means that structures, behaviour and values are shared. Retrieving a shared value or invoking a shared method is made by delegation, which works as follows:

> "When an extension object receives a message, it first attempts to respond to the message using the behaviour stored in its personal part. If the object's personal characteristics are not relevant for answering the message, the object forwards the message on to the prototypes to see if one can respond to the message. This process of

forwarding the message is called delegating the message." [Lieb86]

If we call "client" the object that receives the message and "server" the shared part in which the related slot is found, the key-point of delegation is that while executing the method found in the server, any reference to a variable (see section 5) or any message sent to "self" has to be interpreted as an access to a variable of the client or as a message sending to the client. This property makes it possible for an object to reuse methods that are defined in its shared part.

3.5. Extending L1 with delegation.

3.5.1 Implicit delegation.

We describe in this section the class *ImplicitDelegation,* derived form *BasicProto*, and its related evaluator. The figure 6 shows what can be done with the language associated to the class *ImplicitDelegation*: creation of new objects having parents, method definitions and message sending with implicit delegation.

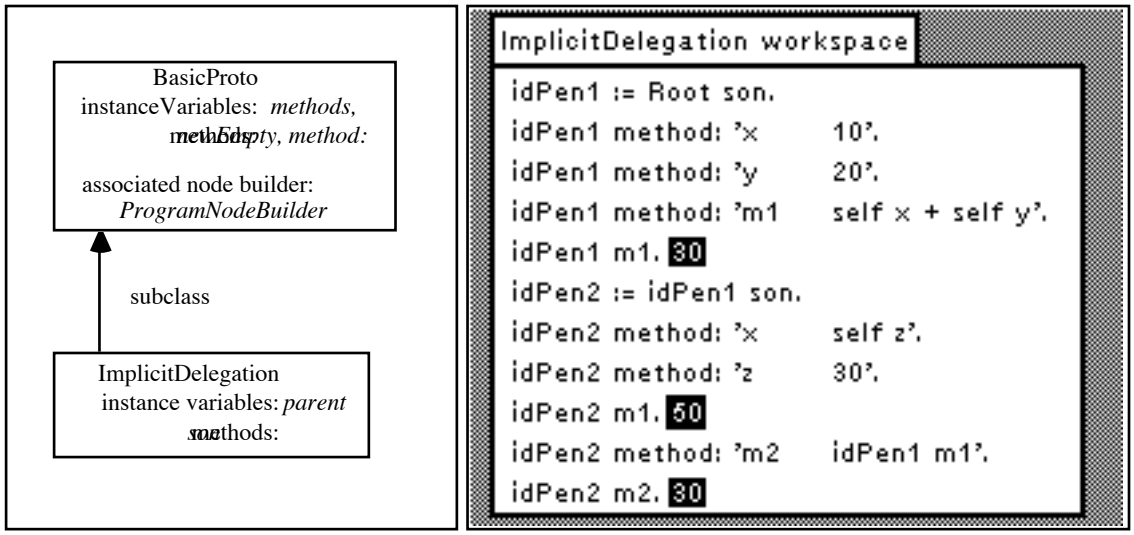


Fig. 6: The class ImplicitDelegation and the related language.

Structure of objects and new primitives

In a system with implicit delegation, objects can have parents. Different choices have to be made concerning the parent link.

The internal representation of the prototypes in our new language will have two fields, one pointing to the parent and one, inherited from *BasicProto* pointing to the methods dictionary. In figure 6, idPen1 is created as a son of Root and idPen2 as a son of idPen1. Sending the message *m1* to *idPen2* returns 50, *m1* has been delegated to the parent and executed in the right environment. Sending the message *m2* to *idPen2* sends the message *m1* to *idPen1* but as it is not a delegation, it returns 30.

The evaluator.

As long as delegation is implicit, the question whether to express it or not via

message sending does not hold; the evaluator is responsible for finding the right method and installing the right context when applying it.

The delegation algorithm is very similar to the way class-based language lookup algorithms follow the superclass link to find an inherited method. The slight difference between delegation and class-inheritance [Naka90] is that with class inheritance, the client and the server are the same object[10] whereas with delegation the client and the server are different. This remarks applies clearly when message sending to the pseudo-variable "super", fundamental for reusability, is to be implemented (see next section). Let us suppose for now that we do not use "super".

With that restriction, the only difference between *ImplicitDelegation* and *BasicProto* evaluators lies in the way methods are searched in the objects. The sequencing of operations and the management of contexts are exactly the same. This means that the above method eval: (figure 3) requires no modifications: examining it shows that the pseudo-variable *self* is bound to *rec* in the new context (line 11) wherever the method is found. Considering the message "self x" in the method m1, after the initial call "idPen2 m1", this is what is needed. Of course, the inherited method *methodNamed:* (figure 3) has to be redefined on *ImplicitDelegation* and is now responsible for finding the method either in the receiver or in its parent considered as the implicit server.

```
methodNamed: name  "defined on ImplicitDelegation"
    if  (self hasMethod: name)
        then    I return (methods at: name)
        else    if  parent equals nil
                    then I return nil
                    else I return (parent methodNamed: name)
```

Sending messages to "super".

The pseudo-variable "super" is conceptually similar to its Smalltalk counterpart or to the Clos function "call-next-method". Adding the following method: "*idPen2 method: 'm1 1 + super m1'* " and sending *m1* to *idPen2* will return 51. Introducing "super" in the implicit delegation mechanism requires modifications in the evaluator that are very similar to those necessary for explicit delegation. Indeed saying "super m1" does not means "send the message m1 to the appropriate object" but "delegates the message the message to the appropriate object while preserving the original client"; the only difference with explicit delegation being that, here, the evaluator is responsible for finding the "appropriate object".

4. Modifying states of objects and ensuring encapsulation.

[10]The client is the reveiver and the server is also the receiver since the method is found in its class or in a superclass of its class, passing form a class to the superclass does not change the object considered as server (this has nothing to do with considering classes as objects ot not).

Up to now, objects variables (states of objects), for example the x-position of a pen, were simulated with methods returning constants values. In the three languages *basic-proto*, *implicit-delegation* and *explicit-delegation* presented in the above sections, objects states cannot be modified simply[11] and there is no encapsulation. Lack of encapsulation means that the internal state of objects are, by default, accessible from anywhere in the system by message sending. The next step towards our simulation of actual prototype-based languages consists in explaining the various solutions for modifying objects states and introducing encapsulation.

4.1. Two techniques for conceptually disjoining variables and methods

Accessing variables by references to their names.

. Any message can be sent from anywhere in the system but a reference to a variable supposes that the variable is visible. The evaluator (or the compiler) owns the visibility rules ensuring encapsulation: variables referenced within methods should be either local variables of the method or variables of the receiver of the applied method or global variables. Encapsulation[12] comes from the impossibility to access objects variables via message sending unless accessor methods are provided. Modifying variables values supposes that a write accessor be part of the interface of its owner and that an assignment instruction in the language (see the method *setY:* in figure 10).

The problem comes with delegation. Without delegation, a variable which is not a temporary or a global cannot be anything else than a variable stored in the current receiver of the method in which the variable is referenced. With delegation, variables values can be inherited and objects can subsequently be split in different parts. For example, if *t1* is a turtle with variables [y -> 30, heading -> 90], with parent a pen *p1* with variables [x ->10, y -> 20], a part of *t1*, the variable *x* and its value, is stored in *p1*. If a method (defined on t1 or on one of its parents) is applied to t1 in which x is referenced, finding x's value will require a lookup in t1's parent. The problem is the same for assignments.

To sum up, distinguishing variables access from message sending gives encapsulation for free but requires that the delegation algorithm be duplicated variables accesses

Having one entity: the slot, with distinct internal representations for variables and methods.

The second solution has been proposed in Self and is a consequence of that last

[11] Defining a method with the same name and returning a different value will destroy the original one.

[12] As far as the language allows to add methods on existing object (as in Smalltalk for classes), it is always possible for a user to add an accessor method for a particular variable but as far as this accessor was not provided by the implementor, the user is warned that this is a potentially dangerous action.

remark: as far as the same dynamic binding algorithm has to be applied to retrieve both state and behaviour scattered in the shared parts of objects, both can be accessed by message sending and have the same external status of slots. The difference between them, and therefore the solution for modifying states, lies in the way slots representing variables and slots representing methods are created. The system is able to recognise, at creation time, whether the slot will hold a variable or a method. As far as a slot representing a variable is to be created, a different internal representation is provided and a related assignment slot[13] (with the same name followed by ":") is automatically created. Having the same mechanism for accessing methods and variables, dynamic binding for variables requires no extra mechanisms.

The problem here comes with encapsulation. As far as a variable's value can be accessed by sending a message, the selector of which being the name of the variable, they become visible and modifiable from anywhere in the system. Achieving encapsulation thus requires additional mechanisms.

4.2. Extending existing classes with variables and encapsulation.

We describe here how we have integrated in our platform languages in which variables and methods are distinguished. We have named the class in which variables are introduced: *Encapsulation*, for, as we have said, the separation of states and behaviour makes variables privates by default. *Encapsulation* is a subclass of *BasicProto* which holds a new instance variable named "*variables"* to store the variable dictionary of objects and a new primitive: *var:value:* to create and initialize variables. The figure 10 illustrates the way variables are defined and used; note the automatic delegation of variables values when the message m1 is send to *idPen2* and how assigning the variable *y* in the method *setY:* owned by *idPen2* also modifies *idPen1*.

Since the interesting problem with the addition of variables occurs when delegation is introduced, we will directly describe the language including the both characteristics. We face here our first problem in designing the taxonomy. The ideal scheme would be to combine by multiple inheritance the classes ImplicitDelegation and Encapsulation (fig. 9a). Because Smalltalk does not support multiple inheritance, we have derived our new class, named *ImplicitDelegationAndEncapsulation* (let us call it *IDAE*) from the class *ImplicitDelegation* and we have duplicated the instance variables and methods of the class *Encapsulation* (fig. 9b).

[13]Except for read-only variables.

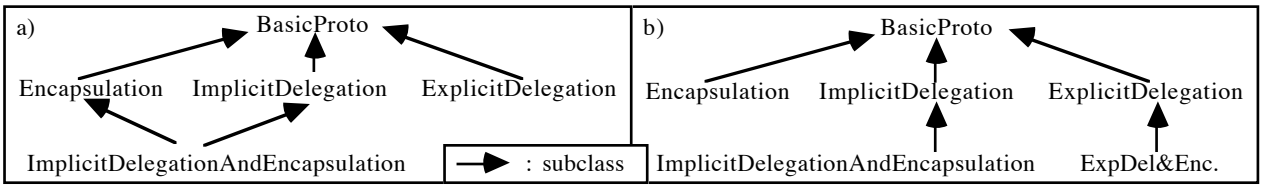


Fig. 9: an ideal taxonomy with multiple inheritance simulated with simple inheritance.

A new program node builder class, let us call it *IDAENodeBuilder*, is associated to the class *IDAE* ; it is a subclass of *ProgramNodeBuilder* and simply states that when a method is parsed and a variable access (resp. a variable assignment) is encountered, an instance of the new class *EncapsulationVariableNode* (resp. *EncapsulationAssignmentNode*) instead of *VariableNode* (resp. *AssignmentNode*) should be created.

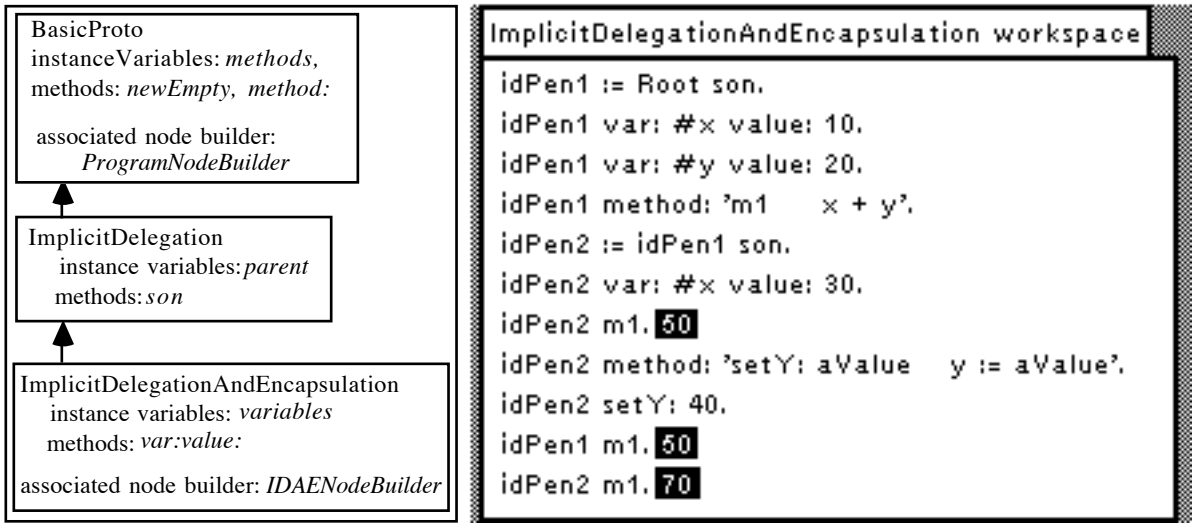


Figure 10: The class ImplicitDelegationAndEncapsulation and the related language.

The difference between the evaluator for this language and the one for the implicit-delegation language, lies in the new method *eval:* defined on the class *EncapsulationVariableNode*. (fig. 11). The methods *variableOwner* and *varValue:* are defined in the classes *Encapsulation* and *IDAE* , the former returns either the owner of the variable, the receiver or one of its parents, the latter directly access the variable dictionary of objects. The method *eval:* on *EncapsulationAssignmentNode* is similar except for the read access to the owner that becomes a write access.

```
eval: context   "defined on EncapsulationVariableNode"
    | client varName owner |
    varName := name asSymbol.
    if   varName is bound in context   "this is either self, or an argument or a temp"
         then    I return (context at: varName)
         else    [client := context at: #self.
                 owner := client variableOwner: varName.
                 if   (owner == nil)   "the variable belongs neither to the receiver nor to its parents"
                      then I return (client globalVarValue: varName)   "is it a global variable?"
                      else  I return (owner varValue: varName)   "the value is asked to the owner"
```

Figure 11: Eval: on EncapsulationVariableNode.

This illustrates how to perform dynamic binding for variables without externally accessing them by message sending, thus giving them the status of private properties.

Finally, extending the class *ExplicitDelegation* raises the crucial problem of how to explicitly express delegation for variables. A first solution consists in redefining variables at the delegating object level and to assign them with a special keyword, known of the system, saying how to delegate it. We have implemented a second solution, not described here, that consist in following the same delegation path than the one used to find the method in which the variable is accessed[14].

5. Existing Systems

The finality of the platform is to easily implement simulations of various prototype-based languages and to write comparative programs. Up to now, we have integrated in the platform simulations of Self, Exemplars, Object-Lisp [Macl] and Act1. The Appendix A is a Smalltalk snapshot showing the current hierarchy of classes representing prototype-based languages. The position in the hierarchy of a class representing each existing language figures out the kind of instructions that can be written, the kind of mechanisms that are available in the language and the operational semantic of these mechanisms. The platform makes it possible to know exactly in each case what happen when a message is sent or when a variable is modified, but of course it gives neither information on the real internal representation of objects nor on how the mechanisms are really implemented in the actual language. Here are some precisions on two of these simulations.

The Self language provides implicit delegation through the parent link of objects and blends variables and methods, as far as most of the messages are send to "self", the syntax allows the receiver to be omitted. The class *SelfLike*[15] thus inherits from *ImplicitDelegation*. A new kind of variable node has been defined for which the method *eval:* interprets symbols in position of variables as message sending to the current value of "self" The new primitives *addSlots:* and *parent:* respectively creates slots in the Self way (cf. section 4) and allows users to dynamically modify the parent of an object. The snapshot gives an example of Self-like code in the "SelfLike Workspace".

The Examplars system is an attempt to separate subtyping from implementation hierarchies. It includes classes and prototypes (named examplars). Examplars behave exactly as instances of the class *ImplicitDelegationAndEncapsulation,* the link to their shared parts being named "superExamplar". They however have an additional link to their class. Classes own general information about exemplars such as their type or the prototypical examples that should be cloned when a new object of the type they represent is to be created. The class *ExemplarsLike* has been created as a

[14]A similar technique is used in Self to deal with potential multiple inheritance conflicts.

[15]The suffix "like" means that the simulation does not pretend reflect all of the possibilities of a language but only the subset of them directly related to the essence of prototype-based programming.

subclass of *ImplicitDelegationAndEncapsulation* and owns a new instance variable to store exemplars classes[16]. The snapshot gives an example of Exemplar-like code in the "SelfLike Workspace".

6. Conclusion

We have given an overview of the Prototalk platform, a framework for the rapid implementation and the use of prototype-based languges interpreters. Those interpreters are built for language understanding and do not pretend to be efficient. The framework architecture makes it very easy to develop an interpreter for a new langage by specialization of the representation of existing ones.

To ease the use of the framework, it would be possible (future works) for a user to choose with a set of radio buttons the characteristics of the prototype-based language he want to test and to have the interpreter almost entirely automatically generated

Prototype-based programming languages are still in their infancy and their underlying concepts await firmer semantic grounds and a more complete understanding of the consequences of alternative mechanisms to implement them. In this paper, we have given the first results of an effort to study, in a systematic way, the primitive concepts of prototypes. We have explored their alternative implementations, using a common platform written in Smalltalk-80 which classifies them through an inheritance hierarchy.

The root of this inheritance hierarchy is a class named BasicProto representing a minimal prototype-based language, which provides only three basic primitives: creating a new empty object, dynamic addition of new slots to existing objects, and message passing. Although very poor, such a language is usable, but it does not provide any way to share common behavior and/or state values. To go beyond that, we have explored two other primitives: cloning and delegation. Cloning, when interpreted as shallow copying of an existing object, exhibit a form of sharing we have called the sharing of values, where both the model object's and the clone's slots point to the same objects just after the cloning and until one of them change the contents of its slots. Delegation, when interpreted as implicit delegation through a parent link, exhibits another form of sharing we have called the sharing of slots, where a child object always shares the slots of its parent for which it delegates messages (it cannot answer itself), thus activating them. After introducing these two

[16]Describing how exemplars classes have been represented is beyond the scope of this paper.

primitives, we have shown that they are both needed in a prototype-based language, because they achieve very different forms of sharing. Implementations of delegation has been proposed, and several alternatives explored. Explicit delegation has been discussed and implemented in our platform.

The last issue we have addressed is state representation, state changes and encapsulation. Two main alternatives are actually proposed: a unique concept of slots melding object variables and methods, as suggested by Self, or, on the contrary, distinct object variables and methods. We have shown that, in the first case, the late binding of variables, needed to implement delegation comes for free since they are accessed in the same way as methods, but that in the latter case, it is encapsulation of objects that comes for free, because accessing a variable cannot be done from the exterior of objects, but that the late binding of variables must then be implemented. Finally, we have compared some existing languages using their respective position in the class hierarchy of our implementation.

An important outcome of this paper is a discussion of the properties of a prototype-based language having the three important primitives for the creation of new objects from existing ones: dynamic addition of slots, cloning and delegation. We enlightened in Section 3.5 the potential complexity of software development in such a language, because of the rapid growth in the number of alternative ways to create a new object from existing ones as an application scales up. In the future we want to look at new models and new programming methodologies to help programmers in choosing the appropriate way to define such new objects, and to insure the reusability of objects in presence of the complex relationships existing among them. The methodology of traits [UCCH91] is a first tentative in towards a more disciplined world of prototypes, but it raises important semantic questions (see §3.2). Whether or not such methodologies go against the basic assumptions of prototypes is also an open question.

References

[Born81] A.H. Borning. The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory. ACM Transaction on Programming Languages and Systems, 3(4):353-387, October 1981.

[Born86] A.H. Borning. Classes versus Prototypes in Object-Oriented Languages. In Proceedings of the IEEE/ACM Fall Joint Conference, pages 36-40, 1986.

[BoWi85] D.G. Bobrow and T. Winograd. An Overview of KRL, a Knowledge Representation Language. In R.J. Brachman and H.J. Levesque, editors, Readings in Knowledge Representation, chapter 13, pages 263-285. Morgan Kaufmann Publishers, 1985. (originally published in Cognitive Science 1, 1 (1977), pp. 3-46).

[ChUn91] C. Chambers and D. Ungar. Making Pure Object-Oriented Languages Practical. Proceedings of OOPSLA'91, ACM Sigplan Notices, 26(11):1-15, November 1991.

[ChUL89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of Self, a Dynamically-typed Object-Oriented Language Based on Prototypes. Proceedings of OOPSLA'89, ACM Sigplan Notices, 24(10):49-70, October 1989.

[CUCH91] C. Chambers, D. Ungar, B.-W. Chang, and U. Hölzle. Parents are Shared Parts of Objects: Inheritance and Encapsulation in Self. Lisp and Symbolic Computation, (4):207-222, 1991.

[HCCU90] U. Hölzle, B.-W. Chang, C. Chambers, and D. Ungar. The Self Manual, version 1.0. distributed with the Self software release, from Stanford University, July 1990.

[HöCU91] U. Hölze, C. Chambers, and D. Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In Proceedings of ECOOP'91, volume 512 of Lecture Notes in Computer Science, pages 21-38. Springer-Verlag, July 1991.

[KhAb90] S. Khoshafian and R. Abnous. Object Orientation - Concepts, Languages, Databases, User Interfaces. Wiley, 1990.

[LaLo89] W.R. LaLonde. Designing Families of Data Types Using Examplars. ACM Trans. on Prog. Languages and Systems, 11(2):212-248, April 1989.

[Lieberman 81] H.Lieberman :A preview of Act1.AI memo No 625, MIT, June 1981.

[Lieb86] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. Proceedings of OOPSLA'86, ACM Sigplan Notices, 21(11):214-223, November 1986.

[Lieb87] H.Lieberman\ : Reversible Object-Oriented Interpreters. in Proceedings of European Conference on Object-Oriented Programming (ECOOP'87), special issue if BIGRE No 54, pp 13-22, June 1987, Paris.

[LaTP86] W.R. LaLonde, D. Thomas, and J.R. Pugh. An Examplar Based Smalltalk. Proceedings of OOPSLA'86, ACM Sigplan Notices, 21(11):322-330, November 1986.

[Macl] Macintosh Allegro Commo Lisp Reference Manual, version 1.3.

[MGDV90] B.A. Myers, D.A. Giuse, R.B. Dannenberg, B. Vander Sanden, D.S. Kosbie, E. Pervin, A. Mickish, and P. Marchal. Garnet, Comprehensive Support for Graphical, Highly Interactive User Interfaces. IEEE Computer, 23(11):71-85, November 1990.

[Naka90] S.Nakajima : Metalevel issues in a Prototype-based Object-Oriented Programming Language. In Informal Proceedings of the first workshop on reflection and metalevl Architectures in OOPSLA/ECOOP'90, Ottawa, October 1990.

[Smit86] R.Smith. The Alternate Reality Kit: An Animated Environment for Creating Interactive Simulations. Proc. of the 1986 IEEE Computer Society Workshop on Visual Languages, Dallas, Texas, pages 99-106, June 1986.

[StLU88] L.A. Stein, H. Lieberman, and D. Ungar. A Shared View of Sharing: The Treaty of Orlando. In W. Kim and F. Lochovosky, editors, Object-Oriented Concepts, Applications and Databases. Addison-Wesley, 1988.

[Stei87] L.A. Stein. Delegation is Inheritance. Proceedings of OOPSLA'87, ACM Sigplan Notices, 22(12):138-146, December 1987.

[Stei89] L.A. Stein. Towards a Unified Method of Sharing in Object Oriented Programming. Department of Computer Science, Brown University, 1989.

[UCCH91] D. Ungar, C. Chambers, B.-W. Chang, and U. Hölzle. Organizing Programs without Classes. Lisp and Symbolic Computation, (4):223-242, 1991.

[UnSm87] D. Ungar and R. Smith. Self: The Power of Simplicity. Proc. of OOPSLA'87, ACM Sigplan Notices, 22(12):227-242, December 1987.

Appendix A

Overview of the Smalltalk-80[17] platform for prototype-based languages simulation.

[17]The platform has been programmed with Objectworks\Smalltalk, releases 2.5 and 4.0; Objectworks\Smalltalk is a trademark of ParcPlace System, Inc.