Prototalk: an Environment for Teaching, Understanding, Designing and Prototyping Object-Oriented Languages

Alexandre Bergel, Christophe Dony, Stéphane Ducasse

▶ **To cite this version:**

Alexandre Bergel, Christophe Dony, Stéphane Ducasse. Prototalk: an Environment for Teaching, Understanding, Designing and Prototyping Object-Oriented Languages. International Smalltalk Conference (ISC'04), Sep 2004, Koethen, Germany. inria-00533452

HAL Id: inria-00533452
https://inria.hal.science/inria-00533452

Submitted on 6 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Prototalk : an Environment for Teaching, Understanding, Designing and Prototyping Object-Oriented Languages.

Alexandre Bergel [a] Christophe Dony [b] Stéphane Ducasse [a]

[a] *Software Composition Group*
Institut für Informatik und angewandte Mathematik
Universität Bern, Switzerland

[b] *LIRMM, Montpellier, France*

Abstract

With prototype-based languages, concretization and abstraction are unified into a single concept a *prototype*. Prototype-based languages are based on a simple set of principles: object-centered representation, dynamic reshape of objects, cloning and possibly message delegation. However, they all differ in the interpretation and combination of these principles. Therefore there is a need to compare and understand.

In this paper we present Prototalk, a research and teaching vehicle to understand, implement and compare prototype-based languages. Prototalk is a framework that offers a predefined set of language data structures and mechanisms that can be composed and extended to generate various prototype-based language interpreters. It presents a classification of languages based on different mechanisms in an operational manner.

1 Introduction

Born at the end of the eighties [1][2][3][4], and influenced by *frame* and *actors* languages, prototype-based languages propose an object-oriented model based on the object as the sole entity of structure and computation. The prototype paradigm and the semantics of its main aspects have been declined into several variations. In the first age of prototype-based languages, research languages such as Self [3][5], Agora [6] [7], Kevo [8], Obliq [9], Garnet [10], Moostrap [11],

Email addresses: bergel@iam.unibe.ch (Alexandre Bergel), dony@lirmm.fr (Christophe Dony), ducasse@iam.unibe.ch (Stéphane Ducasse).

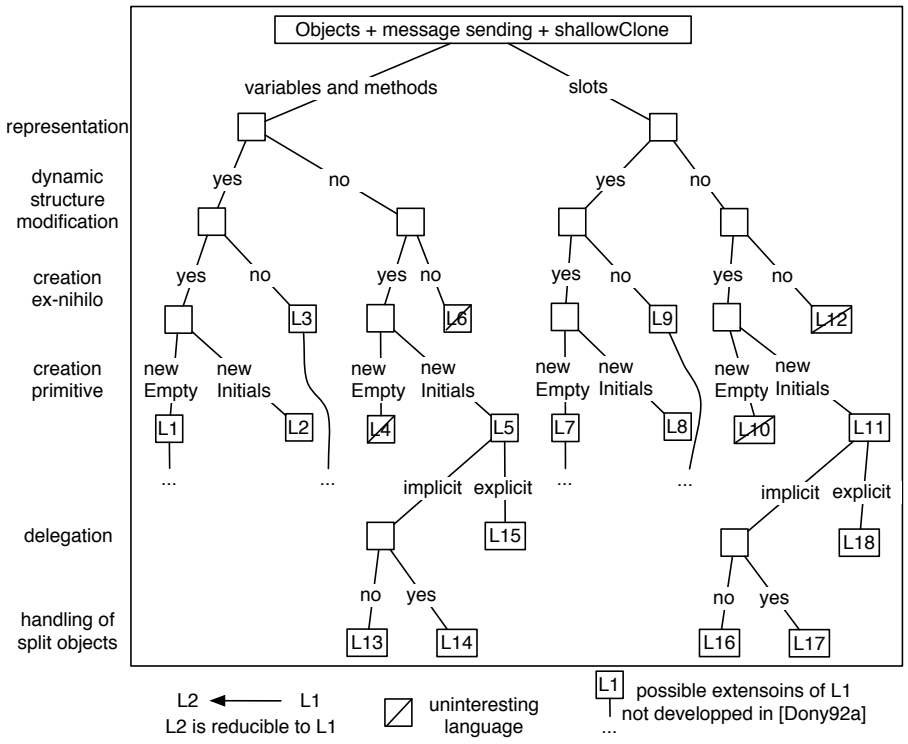


Fig. 1. Classifying prototype-based languages [16]

and ACT-1 [2] [12] have been developed and others such as NewtonScript [13] [14] and GlyphicScript have been used in industrial setting.

While the concept of prototype which results from a desire for simplification and unification seems simple at first glance, prototype-based languages come with different models and mechanisms in key language aspects: knowledge representation, object creation and modification, information sharing, message delegation, method lookup. The Treaty of Orlando was the first attempt at defining more clearly the notion of prototypes and as such acknowledged that understanding the differences and grouping was needed [15]. Later on, various studies have proposed taxonomies and analyses to explain and compare the mechanisms and bring to the fore problems [16] [8] [6] [17] [18] [19]. Figure 1 presents a first taxonomy based on five basic criteria [16]. Most of the languages mentioned in the taxonomy are implemented in Prototalk. Along this paper references to languages named like L1 or L5 refer to the one classified in Figure 1.

Today new prototype-based languages are continuously appearing. Lua [20] is used as a light scripting language. JavaScript with its prototypical instance is still the web-client programming language [21] [22]. More recently IO, a new and extremely compact language with NewtonScript's double inheritance [23], Pic% [24] a new prototype language with mixins, Slate a mix between

Smalltalk and Self [3] and Prothon [25] a mix between Python and Self have been released. New areas such as mobile and distributed applications [26] are discovering the potential benefits of prototype-based languages or of some of the mechanisms or aspects they advocate. Therefore the diversity of prototype-based languages has been and is again quite real. Comparing these languages, being able to design alternative ones, by plugging together adequate aspects, being able to check how far some aspects are compatible is a challenging and useful task since language facets can interact.

The paper presents Prototalk, a framework for implementing interpreters for various prototype-based languages by specializing and composing existing components. The goal of Prototalk is twofold: Firstly, it supports the understanding, classification or teaching of existing languages in a syntactically *uniform* environment. Second it helps designing and testing new languages.

Prototalk follows the approach *"by construction"* promoted by Actalk [27] and ObjVLisp [28] where concurrent languages or reflective class-based ones are decomposed and built from scratch. Like ObjVLisp and Actalk, Prototalk is also an open pedagogical laboratory used to teach object-oriented development. Enabling students compose or design their own language in a concrete way and get an executable language is a pedagogical advantage.

This article is structured as follows. Section 2 presents the goal and the design decision, from the research and pedagogic point of views, that motivated the initial need of a platform to model prototypes-based languages[16]. Section 3 describes the architecture of the framework and shows how the Self and NewtonScript language can be simulated. Section 4 presents the bases for programs interpretation. Section 5 details the step to integrate a new interpreter in the platform. An interpreter for NewtonScript is presented. This language introduced the idea of having two inheritance hierarchies based on two kinds of parent links. Section 6 presents an evaluation of the use of Prototalk.

2 Goals and Design Decisions

While designing the Prototalk platform we had the following goals in mind:

(1) *Uniformity:* The first objective has been to establish an operational semantics for the basic primitives of the basic prototype-based languages and to to unify the languages description to achieve easy comparisons.
(2) *Minimality:* In the tradition of ObjVLisp, Classtalk and Actalk, we wanted through a minimal kernel to express the most general semantics of prototype-based languages,
(3) *Extensibility:* We wanted Prototalk to be a framework so that new lan-

guages simulation can be integrated efficiently and easily.

(4) *Usability:* We didn't want to restrict our system to a semantic model and a raw implementation but we wanted it to be a fully usable environment, including dedicated browsers and inspectors, to achieve actual experiments with prototype-oriented programming.

Prototalk has been initially conceived and used for research purposes but has rapidly been used for pedagogical purposes to teach object-orientation and program interpretation. Its implementation uses all important object-oriented concepts and techniques. It is a framework composed of three main class hierarchies that also uses the Interpreter design pattern. Besides, one of the most interesting points is that it is not so obvious for a student to understand how a class can describe the structure and behavior of objects of a classless world or what distinguishes primitive methods from user-defined methods.

3 The Prototalk Architecture

In Prototalk, a prototype language is represented by an *object model class* which implements all the language primitives and prototype internal representation (See Section 4.2). In addition, the object model class related to a dedicated program node builder which is responsible for emitting the abstract syntax tree of the program (see Figure 2). Programs written in Prototalk are parsed by the Smalltalk parser that generates an abstract syntax tree (AST). The nodes composing this tree are specified by the program node builder associated with the object model. The resulting abstract syntax tree is used to interpret a program following the Interpreter Design Pattern [29].

Prototalk is a framework that can be used at different levels: At the most basic level a user can experiment with languages by combining existing object models provide by Prototalk and existing interpreters. For example, a language L1 making a distinction between the variable and method representation and supporting encapsulation of the variables is created by executing the following expression:

```
(LanguageBuilder new)
   name: #L1
   model: VariableMethodProto
   builder: EncapsulationProgramNodeBuilder
```

The way to define a new language is to (1) define a new object model with its associated program node builder, (2) define new program nodes specifying the semantics of the language, (3) define new primitives and associating them to a language. These tasks are often based on refining or extending existing classes proposed by the framework as shown in Figures 5 and 4 which refers

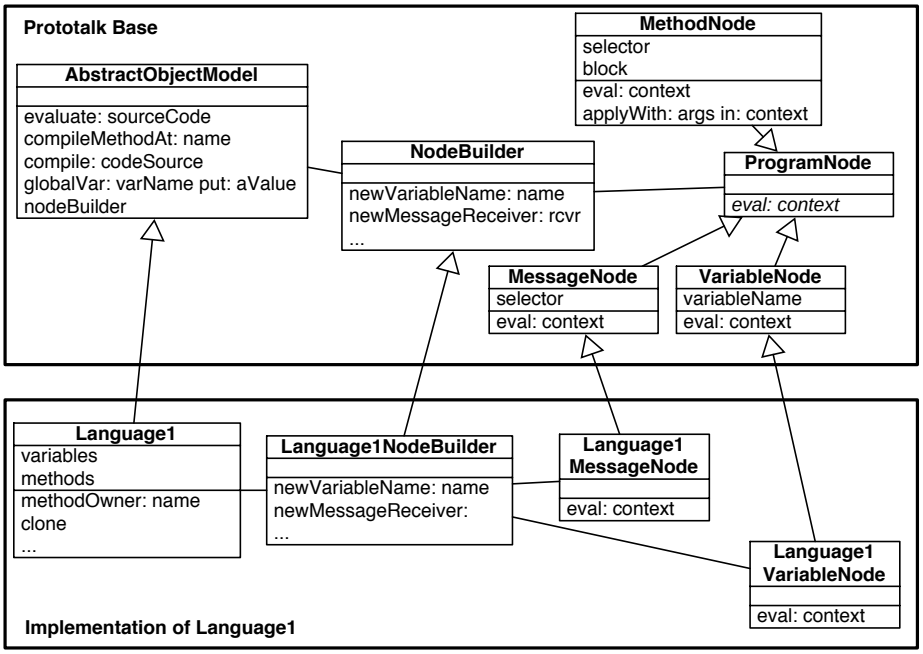


Fig. 2. An object model, its associated program node builder and the corresponding AST elements.

to the taxonomy presented in [16] and in Figure 1. We illustrate these steps in Section 5. Now we present the core elements of the Prototalk architecture.

3.1 Object Model Class

For a given prototype language, the *object model class* has the following responsibilities: (i) it executes programs written in this language by invoking an interpreter on the program nodes, (ii) implements new concepts of the prototype language such as methods or variables lookup, (iii) represents in the Smalltalk memory prototypes created at runtime within a program, and (iv) provides predefined objects such as the empty prototype PRoot or namespace for prototypes.

Executing. A program written in a prototype language is executed by invoking the method evaluate: sourceCode implemented on the class side of the object model class implementing the corresponding language. This method is the starting point of the program interpretation which is based on the Interpreter design pattern [29]. The argument passed to this message is the source code of the program to be executed. The return value of this method execution is the result of the very last statement specified in the program.

The object model class defines all the concepts for a specific prototype language. However, it delegates to the program node builder the responsibility

to assemble a program representation that will then be interpreted (see Section 3.2). A dedicated program node builder is associated with an object model class by the method nodeBuilder. Figure 3 shows a part of the program node builder hierarchy defined in Prototalk.

Prototype Language Primitives. An object model class gathers the different concepts defining the prototype language by implementing primitives related to the following aspects:

- *Object variable and method management.* An object model class has to provide the necessary primitives (i) to add or retrieve methods or variables (*e.g.*, addMethod: code, addVar: name, or addSlot: name) and (ii) to perform some basic object creation operations (*e.g.*, clone or newEmtpy). These primitives are implemented as instances methods defined on the object model class.
- *Method Lookup and Delegation.* Primitives related to message sending and delegation have to be implemented by the object class model through the method methodOwner: name. This method returns the prototype owner of the method name. It defines the lookup semantics. Traditionally it follows the chain of parents objects. This method can be specialized in order to simulate other behaviors such as multiple-parents or double inheritance (Section 5.1).

The method AbstractProto class ≫addPrimitivesOn: defines the primitives that are shared by all the prototype languages. By default any prototype can be cloned, printed and saved in textual format. The method #clone is defined when the structure of the prototype is defined. For example the class SlotProto which unifies variables and methods into slots redefines the primitive #clone. The language L7AndImplicitDelegation which introduces the concept of parent adds two primitives #newSon and #parent.

```
AbstractProto class ≫ addPrimitivesOn: aRoot
    aRoot addPrimitiveNamed: #clone.
    aRoot addPrimitiveNamed: #printOn:.
    aRoot addPrimitiveNamed: #fileIn:.
    ^aRoot

AbstractProto≫ clone
    self subclassResponsibility

L7AndImplicitDelegation class≫ addPrimitivesOn: aRoot
    super addPrimitivesOn: aRoot.
    aRoot addPrimitiveNamed: #newSon.
    aRoot addPrimitiveNamed: #parent:.
    ^aRoot
```

```
SlotProto≫ clone

    | nDic |
    nDic := Dictionary new.
    slots associationsDo: [:each | nDic add: each copy].
    ^ self shallowCopy initMethDic: nDic
```

Prototype Representation. Smalltalk instances of the object model class represent prototype objects. Therefore the object model class defines the common structure of any prototype. For instance prototypes in the SelfLike language are represented as instances of this object model class SelfLike. Proto-

types are Smalltalk objects, therefore Prototalk does not have to deal with the garbage collector.

Predefined Prototypes. Often predefined objects such as the default object to be cloned have to be provided to properly build programs. Usually such a root object offers primitives (cloning, adding variables or methods, ...) accessible to all of its children. For instance the PRoot object is created by the method createRoot defined on the class AbstractProto.

For a given language, PRoot is contained in the global environment and it provides a prototype normally intended to be the root of the object hierarchy.

3.2 Abstract Syntax Tree and Program Node Builder

The method evaluate: sourceCode implemented on the object model class calls the Visualworks Smalltalk parser to generate the abstract syntax tree (AST) of the program source code passed as argument. This AST is composed of nodes obtained from the program node builder associated with object model class. Each language L in our platform is associated with program node builder class that determines, via a set of methods, which kind of interpreter program nodes represent an L expression. This is a hook offered by the Smalltalk compilation framework to specialize the generated AST. The class ProgramNodeBuilder is implemented by the Smalltalk compilation framework. This class emits specific AST nodes when the parser requests it. It offers a common protocol that can be specialized to get different ASTs from the same or different source code. For example, the ProgramNodeBuilder defines the following methods: declareVariableName:, newCascadeReceiver:messages:, newAssignmentVariable:value:, newMessageReceiver:selector:arguments: or newSelfMessageSelector:.

Syntax. In Prototalk, programs are written using a Smalltalk syntax (cf. Section 6). Therefore syntactically, a Prototalk program is a valid Smalltalk one. However, the semantics are different, variable accesses and method invocations depend on the semantics of the language implemented. Most of the nodes composing a prototalk program AST are the Smalltalk ones, except for a few of them that need to be specialized to reflect prototype languages features such as MessageNode and SuperMessageNode for method lookup and VariableNode and AssignmentNode that represent variables.

3.3 Object Model Class Hierarchies

Prototalk heavily uses inheritance to structure and reuse the various aspects of prototype-based languages. Figure 4 gives the hierarchy of object model

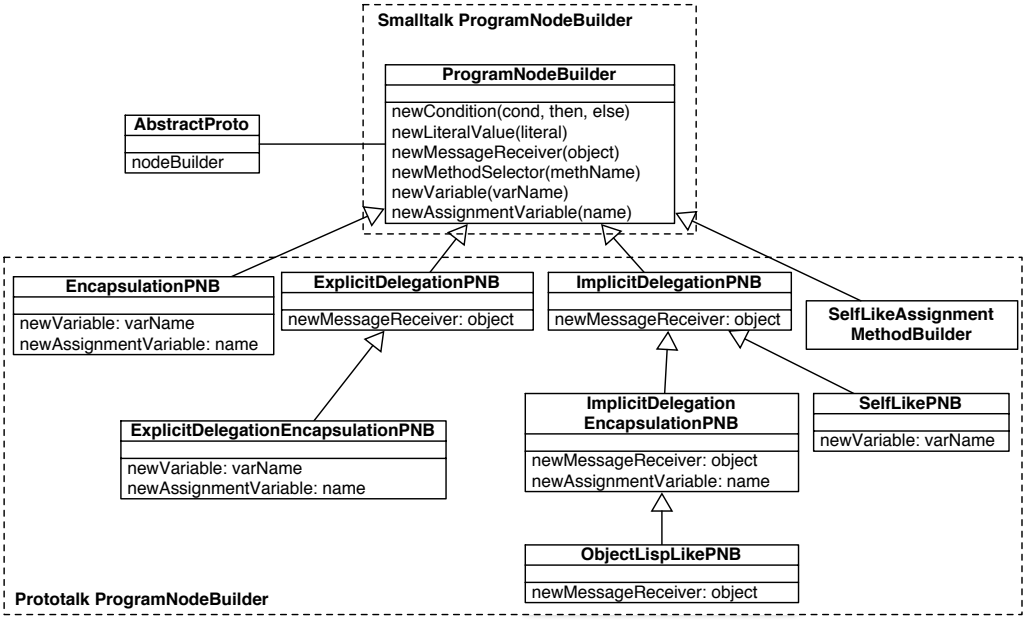


Fig. 3. Hierarchy of the program node builder.

classes representing languages in which prototype objects are defined by a set of slots. For example, the class SlotProto has an instance variable slots and some primitive methods to add new slots (addSlot: name), and to retrieve slots values (slotNamed: name).

Having a class hierarchy for the object models facilitates the extensibility. The definition of a new language is done by simply defining the differences with a previous language. For example the class L11 is a refinement of the class SlotProto that allows prototypes to be created and initialized with a set of slots (method newInitials: slots). The language L11 is refined to the language L11AndImplicitDelegation. The object model class adds an instance variable parent and implements methodOwner: name, parent and newSon. This language is specified by defining the differences with L11.

The object model class hierarchy that defines languages having variables and methods is shown in Figure 5. The class VariableMethodProto defines two instance variables: variables and methods. These variables are initialized in the constructor of this class to an empty set. It also contains all the necessary primitives to manipulate them properly (*e.g.*, addVar: name and addMethod: code to add a variable or a methods). Each prototype objects of this language has its own set of variables and methods.

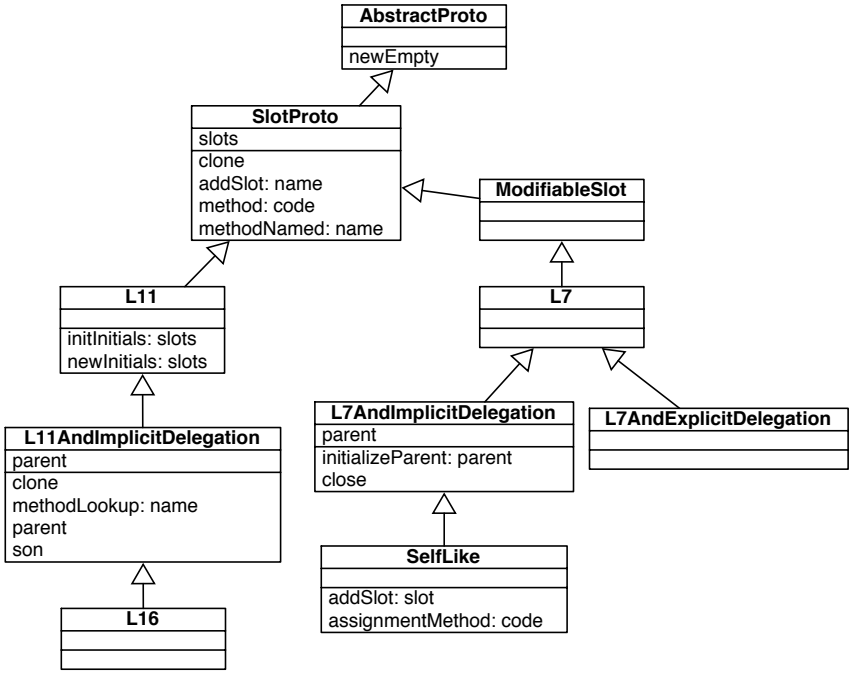


Fig. 4. Hierarchy of object model classes for languages having slots.

4 Basics for Language Interpretation

In Prototalk programs are not compiled, instead interpreted directly from the AST. This AST implements the Interpreter design pattern [29]: each node of the hierarchy is evaluated (through a method eval: context). The interpretation (*i.e.*, execution of a program) starts in its root node. This method is triggered by the object model class when the method evaluate: sourceCode is invoked. Evaluating a node requires a context containing information related to the action performed. The context refers to the receiver of the message currently sent (self), possibly the super one, the variables passed as argument within a method. Access to the self reference is necessary when new variables are defined (*i.e.*, method eval: context of class VariableNode and its subclasses), when new slots are added (methods addSlot: code in the class SelfLike), or when message are sent (method eval: context class MessageNode and its subclasses).

4.1 Variable and Message Nodes

Modeling prototype-based languages has to deal with object representation and message lookup semantics. This means that embedding a prototype language within a class-based language requires the redefinition of the semantics of messages passing and variables lookup. In Prototalk, the AST used to represent prototype program is quite similar to the one offered by Smalltalk except

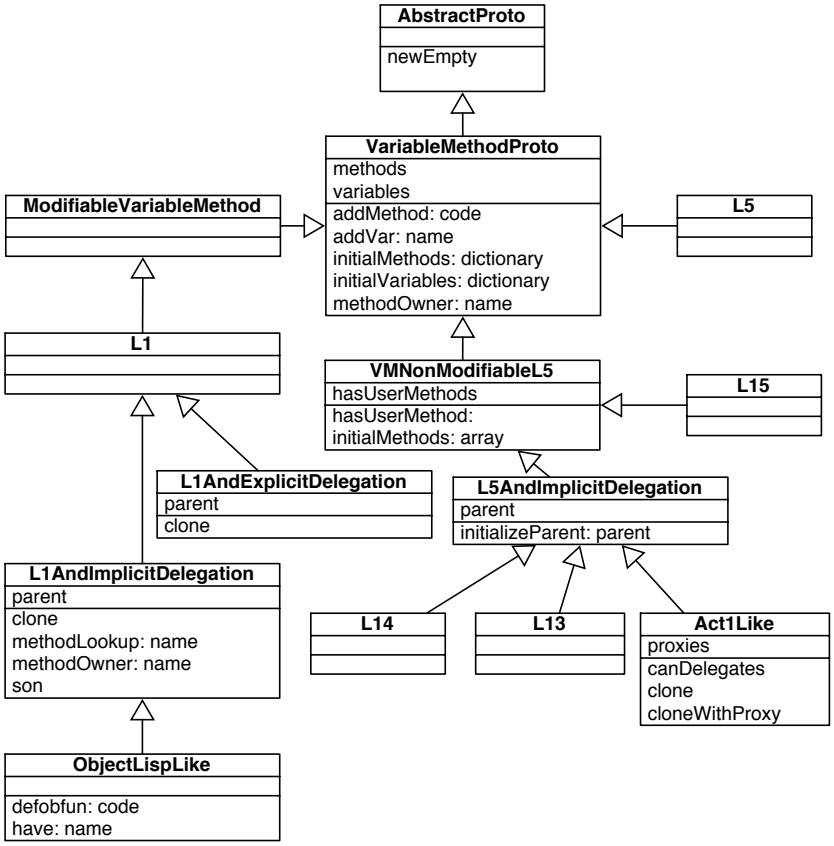


Fig. 5. Hierarchy of object model classes for languages having variables and methods.

that the nodes MessageNode, AssignmentNode and VariableNode have several subclasses to represent the different semantics for sending messages and manipulating a prototype state. Four different message send semantics are implemented in Prototalk: (i) explicit delegation to represent languages like ACT-1 where the delegation has to be done manually (no parent relationship), (ii) implicit delegation to the parents for messages that are not understood by the receiver (*e.g.*, Self), (iii) invocation of a message by invoking the function *ask(receiver, messageName, arguments)* (à la ObjectLisp) and (iv) super call used to explicitly forward a message to the object contained in the *super* slot. Figure 6 shows the specialization of the Smalltalk AST to allow Prototalk to use different language semantics.

4.2 Primitives and Basic Behavior

Primitives refer to the methods implemented on the object model class that can be invoked (by a direct or indirect method call) from within code written in the prototype language implemented by the object model class. While the object model class instance represents prototype, methods defined by the user

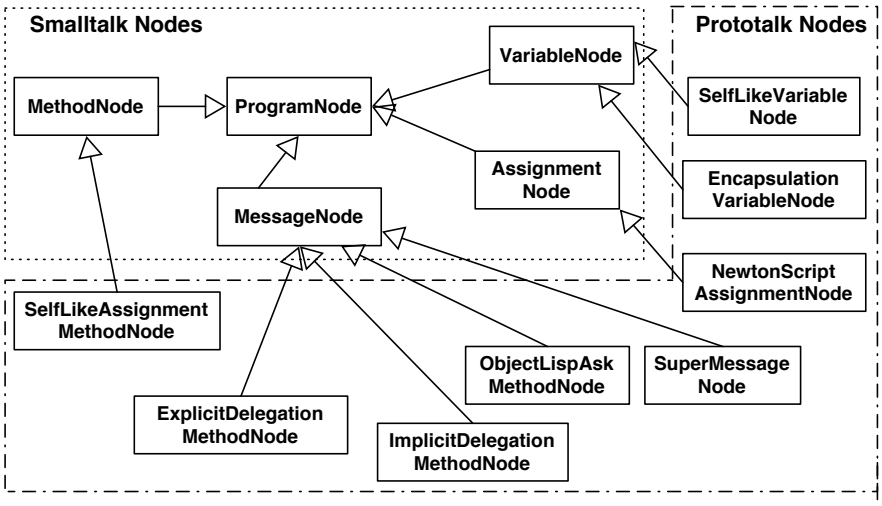


Fig. 6. Specialization of Smalltalk nodes to simulate prototype language behavior.

on these instances are not defined on the object model class. Such methods are stored either in the methods dictionary of an object defined by the class VariableMethodProto or on slot dictionary of an object defined by the class SlotProto (see Figures 5 and 4). These two method holders contain Smalltalk compiled methods in case of primitives, or AST in case of user defined methods.

Some utility methods are provided to enable the programmer to easily manipulate methods such as the method Object≫lookup: name that returns the class implementing a particular method name. In addition the method compiledMethodAt: has been specialized to take care of the fact that the owner of a method is now an object instead of a class (*i.e.*, Smalltalk).

4.3 Default Primitives

The class AbstractProto, which is the root of the object model classes, offers severals methods that can be used as primitives in the prototype-based languages. This class offers eleven primitives.

Four primitives are defined on the instance side of the class AbstractProto. (i) New primitives can be added using addPrimitive: aCompiledMethod named: sel. However it is declared as abstract but it is redefined in its two direct subclasses VariableMethodProto and SlotProto: either to store a primitive in a method or a slot. Global environment for a language can be accessed through a prototype (ii) globalVar: varName put: aValue and (iii) globalVarValue: varName are used for that. (iv) Creation from ex-nihilo (*i.e.*, empty object) is available with newEmpty.

On the class side of AbstractProto seven primitives are offered. Even if these methods are usually not directly invoked during the interpretation of a program they are essential. (i) New primitives are added to the default objects offered to the user (*e.g.*, PRoot) using addPrimitivesOn: aRoot. (ii) New method are added by the user through compile: aString (*e.g.*, this method is indirectly called by addSlot:). (iii) The definition of the PRoot object is contained in the createRoot method. (iv) Compilation and execution of a program is done with evaluate: aStream. The result of this method is the result of the very last statement defined in the program passed as a Stream. The global environment for a language is managed by (v) globalVar: varName put: aValue and (vi) globalVarValue: varName. These methods are invoked by the corresponding method on the instance side. And the reference to the node builder is got by invoking (vii) nodeBuilder.

4.4 Evaluation

Even if programs expressed within Prototalk have a particular semantics, their syntax is always the one of Smalltalk (Section 6). Prototalk extends all of the Smalltalk AST nodes with the bare minimum, that defines a prototype language where: (i) variables do not need to be declared (they are created when first referenced) and (ii) any node of the AST can be particularized using subclassing.

The following classes implement an eval: context methods that simulate the normal Smalltalk behavior (which is identical to Prototalk's *i.e.*, control flow structure): ArithmeticLoopNode, BlockNode, CascadeNode, ConditionalNode, LiteralNode, LoopNode, MessageNode, ReturnNode, SequenceNode.

Differences between Prototalk and Smalltalk only rely on a few nodes points: variable assignment, sending messages, and accessing variables.

As described below, an assignment refers either to binding a new value to a key existing in the context (*i.e.*, self, super, or arguments passed to a method or a block) or creating a new global variable (explained later in this section). Accessing a variable looks at a binding up either in the current context or in the global environment.

```
AssignmentNode≫ eval: context
    | varName val client  |
    varName := variable name asSymbol.
    val := value eval: context.
    (context hasLocalVar: varName)
        ifTrue: [ ^ context at: varName put: val ]
        ifFalse:
            [ client := context at: #self.
            ^ client globalVar: varName put: val ]
```

```
VariableNode≫ eval: context
    | client varName |
    varName := name asSymbol.
    ^ context at: varName
        ifAbsent:
            [ client := context at: #self.
            client globalVarValue: varName ]
```

The class SimpleMessageNode is the node that represents a message send. It implements the default message send semantics. Its eval: context method performs the following steps: (i) evaluation of the receiver in the local context, (ii) evaluation of the arguments related to this message (the method evalList: is implemented on the class Collection), (iii) lookup of the prototype that defines the looked up method and if it is not found then raise an error, (iv) if the method retrieved is a CompiledMethod (*i.e.*, if it is a primitive or if it correspond to a Smalltalk message) then the result of the methods is its execution [1] else, (v) the retrieved method is an AST (it corresponds to a method defined by the user) and it is applied to the arguments using a new context where the receiver is bound to #self.

```
SimpleMessageNode≫ eval: context
    | method rec args newContext owner |
    rec := receiver eval: context.
    args := arguments evalList: context.
    owner := rec lookup: selector.
    owner isNil ifTrue: [AbstractProto unknownMethod raiseWith: selector].
    method := owner compiledMethodAt: selector.
    (method isKindOf: CompiledMethod)
        ifTrue: ["this is either a call to a primitive or a smalltalk message"
            ^ method valueWithReceiver: rec arguments: args ].
    "this is a user-defined method"
    newContext := PContext new.
    newContext at: #self put: rec.
    ^ method applyWith: args in: newContext
```

4.5 Application

The classes BlockNode (representing a Smalltalk block closure) and MethodNode (representing a method definition) implement a method applyWith: args in: context invoked when a block or a method has to be evaluated (cf. last example in the previous paragraph).

Applying a block on a set of arguments (i) extends the local context with the arguments, and (ii) evaluate the block's body.

```
BlockNode≫applyWith: args in: context
  1 to: arguments size do: [:i |
                context at: ((arguments at: i) variable name asSymbol)
                        put: (args at: i)].
  ^ body eval: context.
```

If no error occurs during the application of a method then the result is the application of the related block defining a method. In Smalltalk the body of a method is represented as a block closure referenced by the instance variable block.

```
MethodNode≫applyWith: args in: context
```

[1] note that the context is discarded because the method is not interpreted by prototalk

```
^ AbstractProto returnFromSignal
        handle: [:ex | ex returnWith: ex parameter]
        do: [block applyWith: args in: context]
```

4.6 Extensions in basic classes

The classes Object and Behavior offered by Smalltalk has to be extended in order to make them normal smalltalk objects able to receive message from prototypes. The goal is to make Smalltalk objects usable in any language expressed in Prototalk.

Object is extended with a method lookup: selectorName used to return the class contained in the class hierarchy of the receiver that defines a method named selectorName. Methods are retrieved with methodNamed: name.

```
Object≫ lookup: selectorName
    ^ self methodOwner: selector
```

```
Object≫ methodOwner: selectorName
    ^ self class
        smalltalkMethodOwner: selectorName
```

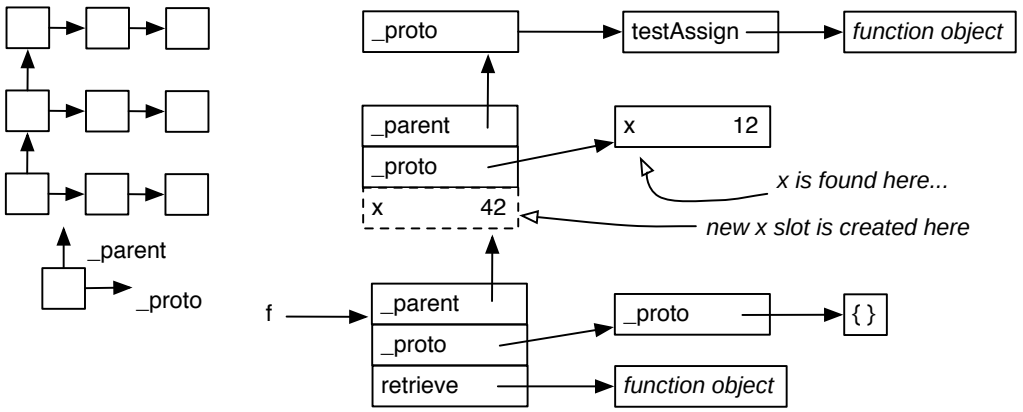
The class Behavior is extended with two methods useful to retrieve a class contained in the super chain of the receiver that defines a method (smalltalkMethodOwner: selector) or a variable (smalltalkVariableOwner: selector).

```
Behavior≫smalltalkMethodOwner: selector
    "Answer the class owner of the method or nil."
    (self includesSelector: selector) ifTrue: [^ self].
    superclass == nil ifTrue: [^nil].
    ^superclass smalltalkMethodOwner: selector
```

```
Behavior≫smalltalkVariableOwner: selector
    "Answer self if the variable is one the receiver variables
     or one of its superclasses variables and nil otherwise."
     (self allInstVarNames includes: selector)
         ifFalse: [^nil].
```

5 Implementing NewtonScript's Double Inheritance Semantics

Prototalk served as a basis to implement the key features of prototype-based languages as discussed. In this section we present how double inheritance of NewtonScript is implemented in Prototalk. We choose NewtonScript since it was one of the few prototype-based language used in production for the Apple Newton PDA [14]. The major design goals of the language were to minimize memory consumption and support graphical user interface definition by introducing an environmental acquisition mechanism [14] [30] that allows sharing between composite graphical objects and their parts.




```
f := {retrieve: func () x, _proto: {_proto: {}}};
f._parent := {_proto: {x: 12}};
f._parent._parent := {_proto: {testAssign: func () x := 42}};
f:testAssign();
Print (f:retrieve());
```

Fig. 7. NewtonScript double inheritance method lookup and variable access.

5.1 NewtonScript

In NewtonScript an object is called a *frame*. A frame element is a slot which can either be an attribute or a function reference. Slots can be inserted or deleted dynamically at run time. New frames can be created from ex-nihilo (empty frame), by cloning and by extending an existing frame.

NewtonScript has a double inheritance scheme in which a prototype link (_proto) is searched prior to a parent one forming a comb-like graph traversal as shown in Figure 7. This specific inheritance supports the building of user-interfaces as well as the minimization of memory [31]. Each frame has a prototype frame from which it is derived (refers to by a slot called _proto). Each time the NewtonScript interpreter does not find a slot variable or function locally it looks into the frame's prototype and then recursively into its prototype until the whole proto chain is searched through. When the slot is not found in the prototype chain of the receiver, it is looked in the parent (following the _parent slot) and the parent prototype chain.

The semantics of assignment is slightly different and supports space consumption minimization: the same lookup occurs to locate the frame containing the variable, but the assignment is only allowed to alter an object in the _parent chain, not the one in a _proto chain. If necessary, a new slot is created in the object on the _parent chain nearest the object where the slot was found as shown in Figure 7. It provides a form on copy-on-write in which the initial value of for a slot comes from the prototype. This way the allocation of the slot to hold a new value is delayed until a new value is actually assigned.

Here is an example of NewtonScript code and its implementation in Prototalk. Even if syntactically the two programs are different, their execution leads to the same objects and construction.

```
Point := {
        x : 0,
        y : 0,
        move: func (newX, newY)
                        begin
                                x := newX;
                                y := newY;
                         end,
        same : func (pt)
                        begin
                         (pt.x = x) and (pt.y = y)
                         end
};
```

```
Point := PRoot clone.
Point addSlot: 'x = 0'.
Point addSlot: 'y = 0'.
Point addSlot: 'moveToX: newX andY: newY
                                x := newX.
                                y := newY.'.
Point addSlot: 'same: p
                        (p x = x) & (p y = y)'.
```

5.2 In Prototalk

The first step is to create the object model class NewtonScriptLike as shown below. Because of the similarity between Self and NewtonScript (implicit delegation, reference to a parent object, slot management, ...) the object model class inherits from the class SelfLike. In NewtonScript a frame has a parent (instance variable inherited from SelfLike) and a prototype named _proto instance variable which has to be defined in the object class model.

```
Smalltalk defineClass: #NewtonScriptLike
        superclass:  #{Smalltalk.SelfLike}
        instanceVariableNames: 'proto '
```

The method methodOwner has to return the object that implement the looked up method. The NewtonScript's method lookup semantics described previously is specified by specializing the method methodOwner: name as follow: if the current frame provides an implementation of a name method then return this frame, otherwise check if one of the frame's prototypes implements it. If not then we lookup in the frame's parent. The lookup in the prototype chain is simply done by running over it: for each frame it is checked if it implements a method named name. Note that this lookup does not have to run over the parent link.

```
NewtonScriptLike≫methodOwner: name
  "return the owner of the method or nil"

 ^ (self includesSelector: name)
       ifTrue: [ self ]
       ifFalse: [ | t |
          t := self methodOwnerInProtoChain: name.
          t isNil
             ifTrue: [
                parent isNil
                  ifTrue: [ nil ]
                 ifFalse: [ parent methodOwner: name ]]
             ifFalse: [ t ]]
```

```
NewtonScriptLike≫methodOwnerInProtoChain: name

 ^ (slots includesKey: name)
       ifTrue: [ self ]
       ifFalse: [
          proto isNil
             ifTrue: [ nil ]
             ifFalse: [ proto methodOwner
                              InProtoChain: name ]]
```

The class NewtonScriptLike has to provide an accessor and a mutator for its instance variable proto (code not shown). These have to be accessible from a program intended to be interpreted. Therefore two primitives have to be added to the PRoot object which is the first prototype from which others can be cloned. By having the primitives proto and proto: defined in PRoot, they are accessible from each object.

```
NewtonScriptLike class≫addPrimitivesOn: aRoot
      super addPrimitivesOn: aRoot.
      aRoot addPrimitiveNamed: #proto.
      aRoot addPrimitiveNamed: #proto:.
      ^ aRoot
```

The object model class has to implement a method nodeBuilder that returns the *class* of its associated node builder (NewtonScriptPNB).

```
NewtonScriptLike class≫nodeBuilder
     ^ NewtonScriptPNB
```

Program Node Builder. The class NewtonScriptPNB inherits from ProgramNodeBuilder. It defines the program node builder used to yield proper nodes when requested by the Smalltalk parser. When the Smalltalk parser parses an assignment, the method newAssignmentVariable: var value: val leftArrow: bool is invoked on the program node builder. A proper node representing an assignment has to be emitted.

```
NewtonScriptPNB≫newAssignmentVariable: var value: val leftArrow: bool
     ^ NewtonScriptAssignmentNode new variable: var value: val leftArrow: bool
```

Sending a message has to trigger the method lookup algorithm described previously in the method NewtonScriptLike≫methodOwner:. Here we reuse the behavior of the class ImplicitDelegationMethodNode which implements a lookup of method and the use of super. The method ImplicitDelegationMethodNode≫eval: invokes the methodOwner: that is used to execute properly methods in presence of the parent slot. When invoked by the parser, the method newMessageReceiver: rcvr selector: sel arguments: args creates then an ImplicitDelegationMethodNode node.

```
NewtonScriptPNB≫newMessageReceiver: rcvr selector: sel arguments: args
     ^ ImplicitDelegationMethodNode new
```

```
            receiver: rcvr
            selector: sel
            arguments: args
```

Accessing a variable in NewtonScript has the same meaning as in Self: accessing a variable is equivalent to trigger its accessor. Referencing a variable foo is equivalent to sending the message foo to itself. On that point NewtonScript behaves in the same way as in Self. Therefore we reuse the SelfLikeVariableNode behavior as follows:

```
NewtonScriptDelegationPNB≫newVariableName: nameString
      ^SelfLikeVariableNode new name: nameString
```

Variable Assignment. The class NewtonScriptAssignmentNode is a subclass of AssignmentNode. It implements the methods eval: context that defines semantics of assigning a value to a variable in NewtonScript. This method requires a context as argument containing the references to *self* and to the temporary local variables.

```
NewtonScriptAssignmentNode≫eval: context
      | varName val client  owner |
      varName := variable name asSymbol.
      val := value eval: context.
      ^  (context hasLocalVar: varName)
            ifTrue: [ context at: varName put: val ]
            ifFalse:
                  [ client := context at: #self.
                   self inAMethod
                        ifTrue: [ owner := client methodOwnerInParentChain: varName.
                                    owner isNil
                                          ifTrue: [ client addSlot: (variable name, ' = ', val printString) ]
                                          ifFalse: [ owner addSlot: (variable name, ' = ', val printString) ]]
                         ifFalse: [ client globalVar: varName put: val]]

NewtonScriptLike≫methodOwnerInParentAndProtoChain: slotName
       "Return the first frame in the parent hierarchy that has (or one of its proto)
        a slotName defined"
   (slots includesKey: slotName)
       ifTrue: [ ^ self ]
       ifFalse: [
           proto notNil
               ifTrue: [ (self methodLookupInProtoChain: slotName) notNil
                           ifTrue: [ ^ self ]].
           parent isNil
               ifTrue: [ ^ nil ]
               ifFalse: [ ^ parent methodOwnerInParentAndProtoChain: slotName ]]
```

First the right branch of the assignment (designed by the value instance variable) is evaluated using the current context. Then, if the left branch of the assignment, the variable named varName), is already defined in the context, it is a temporary variable. Then its value in it is updated. If not, a new frame variable or a global variable is created regarding if this assignment occurs in a method or not.

If the left branch of the assignment occurs in a method and if varName refers no local or global variable, then varName is related to the state of the current

frame (self). According to the semantic given by NewtonScript a new variable is created in the current frame if none of its prototypes already has a slot named varName. If one of these already defines varName, then its value is updated by the value of the assignment.

This implementation of NewtonScript double inheritance shows how already existing semantical elements provided by the framework can be reused or redefined.

6 Evalutation

Prototalk is ideally suited for teaching concepts of object-oriented languages and has been successfully incorporated into the masters degree program at the University of Paris VI, Montpellier and Berne. It served as a tool to help teaching object-oriented languages and software engineering as well as program interpretation. Here a synthesis of what we learned from these experiments.

Prototalk has first proved to be a very interesting pedagogical tool although we did not perform registered and controlled experiments to evaluate and compare its impact. Firstly, it is an obvious help to explain what prototypes are and to ask students to invent or explain various languages. Secondly, its implementation is an interesting laboratory because it uses important object-oriented concepts and techniques. It is primarily a classical framework made of three main class hierarchies (object models, program nodes and program node builder) conceived to be extended. It also uses the Interpreter Design Pattern and extends it in various ways as such it could be called "Interpreter Family" and still remains to be described. Prototalk explicitly separates the object model from the instructions and their associated semantics in a language. Then it uses class specialization and composition to express the fact that various semantics can, in different languages, be associated to a single syntactic construction. For example each subclass of DelegationMessageNode and their respective eval: methods define one semantics of delegation whose study is an excellent way to present the semantics of delegation in prototype-based languages. Another very interesting point is the study of the AbstractProto class and subclasses. This class defines the structure and behavior of classless objects. The study of that apparent paradox (a class been used to describe and represent classless objects) is a key point for the understanding of language implementations and of reflective languages. Methods defined on AbstractProto are primitives of the classless languages. Within such a methods, an instance of VariableMethodProto for example, is a Smalltalk object but the same physical entity is a prototype in the implemented language.

From a research perspective Prototalk has been used to build the taxonomies and the evaluations presented in [16] [18] [19]. Most of important feature of classless languages have been simulated with Prototalk.

Features and Languages Implemented. In addition to the NewtonScript's double inheritance, we modeled in Prototalk the following language facets.

- The proxy link of ACT-1 [2] and its associated explicit delegation mechanism. Cloning was then specialized to also clone also the object proxy.
- ObjectLisp delegation. ObjectLisp, which was a layer on top of Allegro Lisp and presented as a class language, was underneath a prototype language inspired by Lieberan his. In ObjectLisp sending a message was performed using the construct ask. For example the (ask point (have 'y 4)) sends the message have to the object refer to the variable point. The receiver could be any expression, therefore it should be interpreted in the current context, but the arguments are executed in a new environment in which self is bound to the current receiver. The method ObjectLispAskMessageNode≫eval: context makes clear this semantical point.
- Self's dynamic multiple inheritance. Depending on the versions of Self, an object could have multiple parents and their priority was denoted using *. In the current version of Self (4.1), only one parent is used. In addition, the parent can be changed dynamically.
- Examplars [4] mixed classes and prototypes hierarchies. Exemplars was a prospective language that proposed a dichotomy between a subtyping hierarchy made of classes and a reuse hierarchy made of prototypes. In fact Exemplar is very interesting because its classes were an exact intuition of Java interfaces.

```
ObjectLispAskMessageNode≫eval: context
    "context contains self which is the initial receiver in the current delegation
    chain, that is the client and curRec the current receiver."

    | rec newContext result |
    rec := receiver eval: context.
    newContext := PContext new.
    newContext at: #self put: rec.
    arguments do: [:each | result := each eval: newContext].
    ^ result
```

As we already mentioned in Section 3 and illustrated during the implementation of NewtonScript's double inheritance, Prototalk is a framework that provides predefined abstractions that can be reused, extended and combined to create different languages. At a structural level, three different elements are offered: objects with slots, objects with variables and methods, and objects with parent. In addition, object internal representation can be fixed or modifiable. From a behavioral perspective, Prototalk offers 4 different message passing semantics as we explained before, cloning and object creation with initial values are also offered.

About Influence of the Syntax. Prototalk is a platform supporting the coexistence of several prototype languages. Each language has the ability to execute programs written in this language, to debug programs and to test them. Each language expressed can reuse all the facilities provided by the classical Smalltalk environment. Using the Smalltalk syntax has the following advantages:

- Users can experiment with several conceptually different languages without having to learn several syntaxes.
- By removing all the syntactic decorations, conceptual values are clearly offered without being obfuscated.
- To not deal with lexical and syntactical parser keeps the implementation of Prototalk simple and the implementation of new language simple too. This helps to make its kernel focus only on essential mechanisms.

Code expressed in a prototype language has to have a Smalltalk syntax. This means that computation is described by sending messages where a receiver is explicitly designated. For instance a language where messages implicitly sent to self is implementable but the implementer should use a naming convention to distinguish self sends from implicit self-sends à la Self.

7 Conclusion and Further Work

Thanks to new application domains such as web programming, mobile and distributed computing, various kind of programming languages such as scripting, dynamically typed, or object classless languages are subject to a renewal of interest.

In this paper we have described Prototalk, a Smalltalk framework dedicated to the implementation of prototype-based language interpreters. We have presented it as a research as well as a pedagogical tool. Prototalk has been conceived and used in the nineties to evaluate and operationnaly compare into an uniform environment prototype-based languages such as ACT-1, ObjectLisp, Kevo or Self, considered as possible alternatives to class-based languages somehow and sometimes judged too complex or rigid. All the important facets of prototype-based languages are implemented and usable in a syntactically uniform environment that garanties for easy comparisons and classifications.

The paper precisely describes the Prototalk framework architecture based on an extension of the Interpreter design Pattern and its Smalltalk implementation. It explains the value of using an object-oriented programming language was paying off from a research and pedagogical point of view, as it supported well incremental definition and the possibility of reusing what is already exis-

ting. It demonstrates why using Smalltalk, among other advantages, enabled us to easily implement that design pattern due to its fully integrated parser.

The paper illustrates Prototalk reuse capabilities by presenting the integration of the NewtonScript double inheritance semantics in the framework. It also illustrates why Prototalk has been successfully used as a pedagogical tool to teach object-oriented concepts and technologies.

The perspectives of this environment are numerous. Firstly, we plan to integrate in the platform interpreters for those new languages such as IO, Pic%, Slate or Prothon. Secondly, as Smalltalk only supports single inheritance and since our classifications are not only ontological but also operationnal, we faced the tyranny of the dominant decomposition. Indeed we had to follow a main decomposition and sometimes this forces us to duplicate the behavior. Ideally we would have prefer to have well-identified behaviors that we could freely compose together. Investigating the use of traits [32] could be a way to solve this problem and rely less on inheritance to model the languages. More generally, all new techniques which plug together components or aspects could be of interest to improve that work and enable it to evolve. Finally, it appeared to us that designing a similar environment for another family of languages (*e.g.*, frame languages or component-based languages) should not be difficult and that a generalisation is somehow possible.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02) and "Recast: Evolution of Object-Oriented Applications" (SNF 2000-06165-5.00/1). We also thank Gabriela Arévalo and Orla Greevy for their feedback on the paper.

References

[1] A. H. Borning, Classes versus prototypes in object-oriented languages, in: Proceedings of the ACM/IEEE Fall Joint Computer Conference, IEEE Computer Society Press, 1986, pp. 36–40.

[2] H. Lieberman, Using prototypical objects to implement shared behavior in object oriented systems, in: Proceedings OOPSLA '86, Vol. 21, 1986, pp. 214–223.

[3] D. Ungar, R. B. Smith, Self: The power of simplicity, in: Proceedings OOPSLA '87, Vol. 22, 1987, pp. 227–242.

[4] W. R. LaLonde, Designing families of data types using exemplars, Transactions on Programming Languages and Systems 11 (2) (1989) 212–248.

[5] D. Ungar, C. Chambers, B.-W. Chang, U. Holzle, Organizing Programs without Classes, LISP and SYMBOLIC COMPUTATION 4 (3).

[6] P. Steyaert, W. De Meuter, A marriage of class- and object-based inheritance without unwanted children, in: W. Olthoff (Ed.), Proceedings ECOOP '95, Vol. 952 of LNCS, Springer-Verlag, Aarhus, Denmark, 1995, pp. 127–144.

[7] W. De Meuter, Agora: The story of the simplest MOP in the world — or — the sche me of object–orientation, in: Prototype-based Programming, Springer-Verlag, 1998.

[8] A. Taivalsaari, Delegation versus concatenation or cloning is inheritance too, OOPS Messenger 6 (3) (1995) 20–49.

[9] L. Cardelli, A language with distributed scope, Computing Systems 8 (1) (1995) 27–59.

[10] B. Myers, D. Giuse, R. Dannenberg, B. Vander Zanden, D. Kosbie, E. Pervin, A. Mickish, P. Marchal, Garnet: Comprehensive support for graphical highly-interactive user interfaces, IEEE Computer 23 (11) (1990) 71–85.

[11] P. Mulet, J. Malenfant, P. Cointe, Towards a methodology for explicit composition of metaobjects, in: Proceedings of OOPSLA '95, Austin, 1995, pp. 316–330.

[12] H. Lieberman, Delegation and inheritance: Two mechanisms for sharing knowledge in object-oriented systems, Bigre + Globule 48 (1986) 79–89.

[13] W. R.Smith, The newton application architecture, in: Proceedings of the 39th IEEE Computer Society International Conference, 1994, pp. 156–161.

[14] W. Smith, Using a prototype-based language for user interface: The newton project's experience, in: Proceedings of OOPSLA '95, ACM, 1995, pp. 61–73.

[15] L. A. Stein, H. Lieberman, D. Ungar, A shared view of sharing: The treaty of orlando, in: Object-Oriented Concepts, DataBases, and Applications, ACM Press, Addison Wesley, 1989, pp. 31–48.

[16] C. Dony, J. Malenfant, P. Cointe, Prototype-based languages: From a new taxonomy to constructive proposals and their validation, in: Proceedings OOPSLA '92, 1992, pp. 201–217.

[17] J. Malenfant, On the semantic diversity of delegation-based programming languages, in: Proceedings of OOPSLA '95, ACM, Austin, 1995, pp. 215–230.

[18] D. Bardou, C. Dony, Split objects: a disciplined use of delegation within objects, in: Proceedings of OOPSLA '96, 1996, pp. 122–137.

[19] C. Dony, J. Malenfant, D. Bardou, Classification of object-centered languages, in: J. Noble, A. Taivalsaari, I. Moore (Eds.), Prototype-based Programming: Concepts, Languages and Applications, Springer Verlag, 1998, pp. 17–45.

[20] R. Ierusalimschy, L. H. de Figueiredo, W. C. Filho, Lua — an extensible extension language, Software: Practice and Experience 26 (6) (1996) 635–652.

[21] ECMAScript Language Specification, European Computer Machinery Association, 1997.

[22] D. Flanagan, JavaScript: The Definitive Guide, 2nd Edition, O'Reilly & Associates, 1997.

[23] Io home page, http://www.iolanguage.com/.

[24] J. D. Wolfgang De Meuter, Theo D'hondt, Pic% intersecting classes and prototypes, in: Andrei Ershov Fifth International Conference on Perspectives of System Informatics, Siberia, Russia, 2003.

[25] Prothon home page, http://www.prothon.org/.

[26] W. D. M. Jessie Dedecker, Using the prototype-based programming paradigm for structuring mobile applications, in: Workshop: Agent-oriented methodologies. Proceedings of OOPSLA 2002, Seattle, WA USA., 2002.

[27] J.-P. Briot, P. Cointe, Programming with explicit metaclasses in Smalltalk-80, in: Proceedings OOPSLA '89, ACM SIGPLAN Notices, Vol. 24, 1989, pp. 419–432.

[28] P. Cointe, Metaclasses are first class: the objvlisp model, in: Proceedings OOPSLA '87, ACM SIGPLAN Notices, Vol. 22, 1987, pp. 156–167.

[29] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, Reading, Mass., 1995.

[30] J. Gil, D. H. Lorenz, Environmental Acquisition – A new inheritance-like abstraction mechanism, in: Proceedings of OOPSLA'96, 1996, pp. 214–231.

[31] J. Noble, J. Potter, J. Vitek, Flexible alias protection, in: E. Jul (Ed.), Proceedings ECOOP '98, Vol. 1445 of LNCS, Springer-Verlag, Brussels, Belgium, 1998.

[32] N. Schärli, S. Ducasse, O. Nierstrasz, A. Black, Traits: Composable units of behavior, in: Proceedings ECOOP 2003, Vol. 2743 of LNCS, Springer Verlag, 2003, pp. 248–274.