John von Neumann Institute for Computing

Jörg Striegnitz, Kei Davis,
Yannis Smaragdakis (Eds.)

Multiparadigm Programming with Object-Oriented Languages

MPOOL 2002

Proceedings

Central Institute for Applied Mathematics

Publication Series of the John von Neumann Institute for Computing (NIC)
NIC Series Volume 13

John von Neumann Institute for Computing (NIC)

Jörg Striegnitz, Kei Davis, Yannis Smaragdakis (Eds.)

Multiparadigm Programming with Object-Oriented Languages (MPOOL)

2nd International Workshop, 11 June 2002

Malaga, Spain

Proceedings

organized by

John von Neumann Institute for Computing

in cooperation with

Los Alamos National Laboratory, New Mexico, USA

Georgia Institute of Technology, Georgia, USA

NIC Series Volume 13

ISBN 3-00-009099-1

Die Deutsche Bibliothek – CIP-Cataloguing-in-Publication-Data
A catalogue record for this publication is available from Die Deutsche Bibliothek.

Publisher:	NIC-Directors
Distributor:	NIC-Secretariat Research Centre Jülich 52425 Jülich Germany Internet: www.fz-juelich.de/nic
Printer:	Graphische Betriebe, Forschungszentrum Jülich

NIC Series Volume 13
ISBN 3-00-009099-1

Preface

This volume contains the proceedings of the International Workshop on Multiparadigm Programming (MPOOL'02), held in Malaga, Spain on June 11, 2002.

Although the idea of combining programming paradigms and languages goes back to the late sixties, when a lot of research effort was spent on the development and investigation of extensible programming languages, this approach has lost neither its importance, nor its elegance.

Programming languages and paradigms are thought models. Their distinguishing concepts have a great influence on how programmers approach the different stages of the software development process. With respect to software quality, it is therefore desirable to let the problem domain determine the choice of the programming paradigm. Especially for larger software projects, this implies the need for languages and tools that support the simultaneous use of different programming paradigms.

Today the object oriented programming paradigm is dominant and ubiquitously employed for design, implementation and even conceptualization and a huge set of tools has been developed and successfully applied over the last two decades. MPOOL tries to bring together people who are trying to build a bridge from OO-centered tools to a toolset that permits free choice of paradigms.

Last year's MPOOL gave evidence that there exists a larger community working in this emerging area. The extended diversity of topics of this year's workshop shows that there is an ongoing advance in programming languages, tools, concepts and methodologies to support multiparadigm programming.

One of the main goals of the workshop (and the reason of publishing this proceedings volume) is to promote and expose work that combines programming paradigms in the framework of OO languages. Building a consensus about the standard background work, the interesting problems, and the future directions is the way to form a community.

Acknowledgment. We wish to heartily thank all the authors for writing very interesting papers and all the members of the Programme Committee for their invaluable help in compiling the program.

June 2002

Kei Davis
Yannis Smaragdakis
Jörg Striegnitz

Workshop Organizers

Kei Davis, Los Alamos National Laboratories, New Mexico, USA
Yannis Smaragdakis, Georgia Institute of Technology, Georgia, USA
Jörg Striegnitz, John von Neumann Institute for Computing, Germany

Programme Committee

Gerald Baumgartner, Ohio State University, Ohio, USA
Kei Davis, Los Alamos National Laboratory, New Mexico, USA
Jaakko Järvi, Indiana University, Indiana, USA
Peter Van Roy, Universite catholique de Louvain, Belgium
Yannis Smaragdakis, Georgia Institute of Technology, Georgia, USA
Jörg Striegnitz, John von Neumann Institute for Computing, Germany

Table of Contents

Object Programming in a Rule-Based Language with Strategies

Hubert Dubois, Hélène Kirchner

LORIA-Université Nancy 2 & LORIA-CNRS
BP 239
54506 Vandœuvre-lès-Nancy Cedex, France
{Hubert.Dubois|Helene.Kirchner}@loria.fr

Abstract. This paper presents a programming framework that combines the concepts of objects, rules and strategies, built as an extension of the rule-based language with strategies ELAN. This extension is implemented in a reflective way in ELAN itself and relies on the same formal semantics, namely the ρ-calculus.

1 Introduction

Object-based languages and rule-based languages have independently emerged as programming paradigms in the eighties. Languages like Ada [Ros92], Smalltalk-80 [GR83], Eiffel [Mey92] or Java [AG96] are well-known and largely used but often lack of semantical basis. Rule-based systems, initially used in the artificial intelligence community, have gained interest with the development of efficient compilers.

The ELAN system [BCD$^+$00] provides a very general approach to rule-based programming. ELAN offers an environment for specifying and prototyping deduction systems in a language based on rewrite rules controlled by strategies. It gives a natural and simple logical framework for the combination of computation and deduction paradigms. It supports the design of theorem provers, logic programming languages, constraint solvers and decision procedures and offers a modular framework for studying their combination. ELAN has a clear operational semantics based on rewriting logic [BKKR01] and on rewriting calculus [Cir00]. Its implementation involves compiled matching and reduction techniques integrating associative and commutative functions. Non deterministic computations return several results whose enumeration is handled thanks to a few constructors of choice strategies. A strategy language is available to control rewriting. Evaluation of strategies and strategy application is again based on rewriting. However, ELAN lack object oriented features, the notion of states, which provides more structuration, and the ability to define structures sharing the same properties.

More recently, the combination of object-based languages and rule-based languages has proved to be quite relevant to formalize and solve advanced industrial problems that require some form of reasoning. Among many other languages, let us mention three of them, more closely related to our approach: CLAIRE [CL96], Oz [HSW95] and Maude [CDE$^+$00].

CLAIRE is combines objects and propagation rules in the logic programming style for problem solving and combinatorial optimization. Single inheritance and full polymorphism are supported. CLAIRE has been realized to deal with applications with complex data structures with the ability to define rules. CLAIRE rules associate a logical condition with an expression composed by one or two objects: when a condition is evaluated into true, the expression is evaluated for the given objects. Conditions are events that denote an evolution of an entity (creation of an object, modification of an attribute, etc...). Rules can be grouped together but control is not explicit.

Oz is a concurrent object oriented constraint language that offers multiple inheritance, higher-order functions and search combinators. The programming language Oz integrates the paradigms of imperative, functional and concurrent constraint programming in a computational framework. The integration of objects into the programming language is interesting because they have defined a small Oz that can support objects for concurrent constraint programming. Propagation rules are also used in the context of constraint solving.

Maude is a language based on rewriting logic, with several extensions, among which Full Maude where object oriented modules can be defined. It offers the possibility to develop concurrent object systems, or *configurations*, where the current state has a multiset structure of objects and messages, and evolves by application of rules that implement message calls.

Compared to these existing languages, extending ELAN with objects provides several advantages. First of all, the fact that ELAN provides a strategy language and strategy constructors to define control on rules, appears as essential for many applications. Second, the main features that characterize object languages (definition of classes composed of attributes and methods; simple inheritance; method call on objects) can be defined in a reflective way, since the extension is defined in ELAN itself. Finally our last, but maybe more important, point is that the extension has a semantics compatible with the rewriting calculus and is achieved by mapping the theory of objects into the ρ-calculus.

In Section 2, we first present the syntax of the object extension of ELAN, including object modules and rules on objects. The encoding of objects and classes into an algebraic theory is presented in Section 3. This provides the basis for the reflective implementation of the extension in ELAN itself. Section 4 first introduces the rewriting calculus on which the semantics of the object extension is defined. The conclusion gives some further perspectives. An extensive example is developed in Appendix.

2 The language extension

Adding object-oriented features to ELAN was motivated by the need of representing structured data and states and to combine them with rewrite rules and strategies describing their evolution. In this section, after a short presentation of the ELAN system, we define the object-oriented extension of ELAN that consists in declaring special modules that we call OModules (OModule for Object Module) where the user can define the classes that he uses. In each module, attributes and methods are defined for each class; the syntax of these object modules is quite similar to object languages like Smalltalk-80 [GR83] or OCaml [RV98].

2.1 ELAN

The starting point of this work was the ELAN system. We briefly present the features of the language that are used in this paper and the reader can refer to [BCD$^+$00] for further details and examples. In order to have some additional informations, the reader can refer to several articles and presentations of the system[1].

The language is close to the algebraic specification formalism with abstract data types defined by operators and rewrite rules, but provides additional specificities that are worth emphasizing. Three main principles have guided the design of the ELAN language.

- First, the language allows rules to be non-terminating and non-confluent, but then their application has to be controlled. A distinction is made between unlabelled rules for computations, which are required to be confluent and terminating, in order to give a unique result, and labelled rules for deductions, for which no confluence nor termination is required.
- Rules and strategies are first-class objects in the language.
- Application of a rule or a strategy on a term may give zero, one, or several results. This non-determinism related to the production of sets of results is handled by backtracking.

Modules Following the algebraic languages tradition, ELAN is modular. A program is a collection of hierarchically constructed modules together with a request, which is a term to be evaluated in this hierarchy. A module may import already defined modules and this importation may be local (not visible outside the module) or global (visible outside). A module also defines a set of sorts, a list of operators with their types, several lists of rules, classified by the type of their left and right-hand sides, and strategies, also defined by operators and rules.

Predefined modules exist in the ELAN library, such as `bool`, `int`, `ident`, `list[X]`...

Confluent and terminating rules Conditional rewrite rules can be grouped together according to the sort of their left (and right) hand side. For rewrite systems with mutually exclusive conditions in rules, we have confluence and terminating properties. An application of such set of rules on an initial term produces a unique result.

[1] See the Web site of ELAN: *http://elan.loria.fr*

Strategy language A strategy language is provided to express control and derivation tree exploration. A few strategy constructors, similar to those for tactics in proof assistants, are offered and efficiently implemented.

– A labelled rule is a primal strategy. Applying a rule labelled **lab** returns in general a set of results. This primal strategy fails if the set of results is empty.
– Two strategies can be concatenated by the symbol "**;**", i.e. the second strategy is applied on all results of the first one. S_1 **;** S_2 denotes the sequential composition of the two strategies. It fails if either S_1 fails or S_2 fails. Its results are all results of S_1 on which S_2 is applied and gives some results.
– **dk**$(S_1, \ldots, S_n)$ chooses all strategies given in the list of arguments and for each of them returns all its results. This set of results may be empty, in which case the strategy fails.
– **first**$(S_1, \ldots, S_n)$ chooses in the list the first strategy S_i that does not fail, and returns all its results. This strategy may return more than one result, or fails if all sub-strategies S_i fail.
– **first_one**$(S_1, \ldots, S_n)$ chooses in the list the first strategy S_i that does not fail, and returns its first result. This strategy produces at most one result or fails if all sub-strategies fail.
– The strategy **id** is the identity that does nothing but never fails.
– **fail** is the strategy that always fails.
– **repeat***(S) applies repeatedly the strategy S until it fails and returns the results of the last unfailing application. This strategy can never fail (zero application of S is possible) and may return more than one result.

But the user may also design his own strategies. The easiest way to build a strategy is to use the strategy constructors to build strategy terms and to define a new constant operator that denotes this (more or less complex) strategy expression. This gives rise to a class of strategies called elementary strategies. Elementary strategies are defined by unlabelled rules of the form [] $S \ \Rightarrow \ strat$, where S is a constant strategy operator and *strat* a term built on predefined strategy constructors and rule labels, but that does not involve S. The application of a strategy S on a term t is denoted $(S)\ t$.

Rules with local variables and patterns Labelled rules and more generally strategies are always applied at the top position of a term. In order to be able to apply them inside expressions, a more general form of rule with local variables allowing to apply strategies on subterms is allowed in ELAN.

One can also generalize variables to patterns, i.e. terms with variables. We define $\mathcal{T}(\mathcal{F}, \mathcal{X})$ as the set of terms built from a given finite set $\mathcal{F}$ of function symbols and a denumerable set $\mathcal{X}$ of variables and $\mathcal{V}ar(t)$ as the set of variables occurring in a term t. We assume that the reader is familiar with these notations and to get more details, he can refer to the main concepts of general logic [Mes89] and rewriting logic [Mes92].

To summarize, the general form of ELAN rules is actually as follows:

$$[\ell] \;\; l \to r \;\textbf{where}\;\; p_1 := (S_1)c_1 \ldots \;\textbf{where}\;\; p_n := (S_n)c_n$$

- $l, r, p_1, \ldots, p_n, c_1, \ldots, c_n \in \mathcal{T}(\Sigma, \mathcal{X})$,
- $\mathcal{V}ar(p_i) \cap (\mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_{i-1})) = \emptyset$,
- $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_n)$ and
- $\mathcal{V}ar(c_i) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_{i-1})$.

In such expressions, **where** $\mathit{true} := c$ is usually written **if** c. The pattern p_i often reduced to a variable x. S_i may be the identity strategy, which is written $()c_i$.

To apply the rule

$$[\ell] \;\; l \to r \;\textbf{where}\;\; p_1 := (S_1)c_1 \ldots \;\textbf{where}\;\; p_n := (S_n)c_n$$

to a subject t, the matching substitution from l to t ($l\sigma = t$) is successively composed with each matching μ_i from p_i to $(S_i)c_i\sigma\mu_1 \ldots \mu_{i-1}$, for $i = 1, \ldots, n$. To evaluate each $(S)c$, c is first normalized using the unlabelled rules, then one tries to apply a labelled rule according to the strategy S. Choice points are set when there are several results and if at some point the set of results is empty, the system backtracks to the previous choice point. When the rule contains a sequence of matching conditions, failing to satisfy the i-th condition causes a backtracking to the previous one.

Associative Commutative functions Associative and commutative (AC for short) functions introduce an intrinsic non-determinism. Since an AC matching problem can have several solutions, one may want to get all solutions of an AC matching problem and build all possible results of rewriting with these different matching substitutions.

When an ELAN rule has a left-hand side l or a pattern p that contains AC function symbols, AC matching is called and can return several solutions. This provides an additional potentiality of backtracking.

As a consequence of these features, the language allows different programming styles. Functional programs are naturally expressed with confluent and terminating rules, while the backtracking mechanism used to handle several results gives a flavor of logic programming and allows to program non-deterministic computations. The main originality is surely the capability of strategy programming for expressing the control of programs in a declarative way.

Adding objects oriented features to ELAN amounts to enrich the language with OModules and rules on objects, described below.

2.2 General syntax of OModules

To each class definition corresponds a specific OModule. In order to differentiate these modules from ELAN standard ones, a particular syntax for OModules is introduced, quite similar to object languages like Smalltalk-80 or OCaml [RV98].

In an OModule, the following items are successively defined: the attributes composing each object of this class, the methods associated to the class, the imported modules and the inherited classes.

Attributes. Each object is characterized by its attributes. To each attribute is associated a type (or sort) defining the set of values that it can take. An initial value is also specified for each attribute which is used when a new object of the class is created.

Example 1 *Let us consider the class `Point` defining points in a bidimentionnal space. Two attributes are declared in this class: the first one, called `X`, defines the abscissa of any point and the second one, called `Y`, for the ordinate. This class is declared in the following object module:*

```
class Point
attributes X:int = 0
           Y:int = 0
End
```

Any object of this class is created with an abscissa and an ordinate initialized to 0.

Methods. A method is a function that can be applied to a given object called *target object*. This method can modify the target object. Calling a method `m` on an object `o` is denoted by `o.m`. This is what is usually called *message passing*. We distinguish here two kinds of methods.

First, the generic methods are those that are automatically defined by the system. These methods deal with the modification and the access to the value of an attribute and also with the creation of an object of a given class.

Let us consider a class C where an attribute A of sort t_A is defined. Two methods, $GetA$ and $SetA$, are associated to this class and to this attribute.

– The message $o.GetA$ calls the method $GetA$ on the object o. The result of this method call is of sort t_A and is the value associated to the attribute A of the object o.
– The message $o.SetA(V)$ calls the method $SetA$ on the object o with the parameter V of sort t_A. The result is the object o updated by replacing the value of the attribute A by the given value V.

In each class, a specific method called `new` is defined which constructs a new initial object of the class, whose attributes are initialized with default values. This object has access to all methods defined in the class. Any new object of this class is created as a copy (a clone) of the initial object and can of course be modified later on. This technique is used in actor languages like Scheme [ADH+98] or Common Lisp [Ste90].

The second kind of methods are those defined by the user: the user methods. A user method is given by its name, a list of arguments if necessary, the sort of the result, possibly local variables and its body.

In the method definition, local variables must be declared before being used in the body in local assignments.

A method can have parameters defined by a formal name and a sort. The object designed by the method call is a particular implicit parameter: it can be referred to by the keyword **self** in the body of the method. This parameter does not have to be declared in the list of arguments.

Method bodies are composed of different instructions: the method call, the concatenation of several instructions, boolean tests and local assignments. These instructions are illustrated in the next example.

Example 2 *Let us consider the class `PointTranslation` with two attributes `X` and `Y` of sort `int` initialized to `0` and which defines two methods: a first one, `TranslateX`, modifies the value of the attribute `X` by adding the value of a parameter `N` to the old value assigned to `X`. The second one, `Translate`, calls the translation over `X` only if a condition on the values of `X` and `Y` is checked. If this is the case, a value for the local variable `N` is computed to allow the method call of `Translate` parameterized by `N`.*

```
class PointTranslation
attributes X:int = 0
           Y:int = 0
method TranslateX(N:int) for PointTranslation
          <self.SetX(self.GetX + N)>
method Translate for PointTranslation N:int
          <if self.GetX > self.GetY ;
           N := self.GetX - self.GetY ;
           self.TranslateX(N)>
End
```

Importation of modules. Each object module can import others modules which are not object modules but standard ELAN modules where sorts, operators, rules and strategies are defined. An object module may use a library of standard ELAN modules, but is not allowed to import another object module. In this way, these importations are different from the inheritance mechanism of object languages presented below.

Example 3 *Let us consider the class Point defined in the example 1. Now, we enrich it in order to define colored points. The colors of the points are defined in a standard* ELAN *module that is called `color.eln` in which the sort `color` is defined as the sort that enumerates all the possible color values. Let us consider that `Black` is one of these values.*

We introduce the new class of colored points ColoredPoint, where the attribute denoting the color of each point is the attribute `Color`, of sort `color` *initialized with the value* `Black`*, we define the object module ColoredPoint as follows:*

```
class ColoredPoint
imports color
attributes X:int = 0
           Y:int = 0
           Color:color = Black
End
```

Inheritance. An object module can inherit attributes and methods from another class by using the keyword **from** followed by a class name. The inheritance mechanism allows a given class `B`, inheriting a class `A`, to specialize the inherited class: attributes and methods defined in `A` are available, but new attributes and new methods can then be defined too. Methods can also be overridden in order to specialize them. This kind of inheritance is called simple inheritance.

The inherited methods are overloaded with the definition of methods with the same name but their sorts are different according to the associated class. The overloading of methods can also be used to redefine a method. The sort and the body of the method are then different between the two classes. This feature also holds for attributes that can be overloaded by giving another initial value.

Example 4 *Let us consider the example of the class Point defining basic points given in Example 1 and the class ColoredPoint defining colored points given in the Example 3. Instead of completely redefining the class ColoredPoint as before, inheritance can be used to simply define the ColoredPoint class from the Point class as follows:*

```
class ColoredPoint
imports color
from Point
attributes Color:color = Black
End
```

The attributes `X` *and* `Y` *from class Point are inherited, as well as the particular methods* `GetX`*,* `GetY`*,* `SetX` *and* `SetY` *associated to the Point class. Thus, we only have to define the attribute* `Color` *while the associated methods* `GetColor` *and* `SetColor` *are then automatically associated.*

The simple inheritance that is defined here can be compared to simple inheritance in Java but is less powerful than in Smalltalk-80 where inheritance is multiple, which means that a class can inherit several other classes.

The complete syntax of OModules and more examples can be found in [Dub01].

2.3 Rules on objects

An object base represents the set of all objects that live at a given time in the system. Rules are defined to delete, modify or add objects in this object base. An informal presentation is given here.

Rules are of the form:

$$[lab]\ \ O_1 \ldots O_k\ \Rightarrow\ O'_1 \ldots O'_m\ \ [\textbf{if}\ \ t \mid \textbf{where}\ \ l]^*$$

where $O_1, \ldots, O_k, O'_1, \ldots, O'_m$ are objects, t is a boolean term and l a local assignment, that both may involve objects and method calls. This rule, as standard ELAN rules previously presented, can be labelled. The O_i objects of the left-hand side represent the matching conditions of this rule. The rule is applied only if these objects can be found in the object base. Each rule mentions the relevant information on the object base; the context is omitted. The order of objects O_i and O'_j in both sides is not relevant since an AC operator is used for the objects base construction. Each O_i object of the left-hand side has one of two possible forms: the first one, $O_i : ClassName_i :: [Att_1(Value_1)\ ,\ \ldots\ ,\ Att_n(Value_n)]$, corresponds to an object of a given class $ClassName_i$ where few attributes Att_i are specified and the second form, $O_i : ClassName_i$, corresponds to an object for which only the class $ClassName_i$ is specified.

In general, the application of a rule on the data base of objects may return several results, for instance when several objects or multisets of objects match its left-hand side. This introduces some non-determinism and the need to control rule application. This is why the concept of strategy is introduced. Strategies are used to control the application of rules on objects: strategies provide the capability to define a sequential application of rules; to make choices between several rules that can be applied; to iterate rules; etc.

Including objects in rules controlled by strategies is now possible in the defined formalism of rules working on an object base. Planification or scheduling problems are easily expressed as shown in [DK00b] where the formalism of rules presented in this paper is enriched in order to also control a constraint base. Strategies are used in both cases to control the application of the rules. Other examples of applications can be found in [Dub01].

3 An algebraic encoding of objects

Our purpose now is to show how the definition of classes in the object modules can be implemented in a first-order algebraic language such as ELAN. We choose here an approach where objects and classes are represented as objects like in the class-based language Smalltalk-80 where the unique entity is the object. This implementation has been guided by a few choices.

There is a distinction between attributes and methods. In many applications, using objects essentially consists in reading or modifying the values of attributes. Thus, we have to define a structure where the values of attributes are quickly accessible. The difference between attributes and methods is a good way to distinguish what represents the state of an object at a given time (the attributes and their associated values) and the functions that can be applied to this object (the methods). However, we do not want to separate an object with its attributes from the set of methods that can be applied to the object. We thus have to define a structure where each object includes the methods associated to it.

The application of a method is defined with rewrite rules. This imposes some (quite reasonable) restrictions on the definition of methods and is compatible with the distinction between attributes and methods in the following way: to an attribute is associated a value which can be modified during the evaluation, but no rewrite rule. An attribute is mutable by rewriting, and complex expressions may be considered as values of attributes. On the contrary, methods are defined by rewrite rules and are non-mutable, i.e. once rewrite rules associated to methods have been defined, one cannot delete or even change them. Thus, an object is composed of mutable attributes, whose values can be changed during evaluation, and of references to non-mutable methods. The reference to any method can be hidden or revealed during the execution and thus, the ability for an object to execute a method can change.

3.1 A representation based on operators and rules

Each object composed of attributes and methods which the object has access to is represented by a term; methods are represented by functions defined by rewriting rules.

– In order to represent the attributes and their corresponding values, the object structure involves a set of pairs *(attribute,value)*. Each attribute is denoted by a constant and each value by a term with the same sort.
– The object structure also involves a set of constants denoting methods. To each of them, an operator is associated that has the same name. To each method body corresponds a rewrite rule right-hand side, with, possibly, local assignments and boolean tests.

This object representation is schematized in Figure 1.

Definition of operators and rules associated to the object representation are detailed in Section 3.2 and in Section 3.3. Operators and rules associated to user methods are detailed in Section 3.4.

3.2 Operators associated to the representation

The signature of operators that are defined to build the representation of objects is as follows:

$$\begin{array}{lll l}
[\_] & : Methods & \mapsto Object & \\
\_,\_ & : Methods \times Methods & \mapsto Methods & (AC) \\
\_ & : Method & \mapsto Methods & \\
\_(\_) & : MName \times MBody & \mapsto Method & \\
\_ & : MName & \mapsto Method &
\end{array}$$

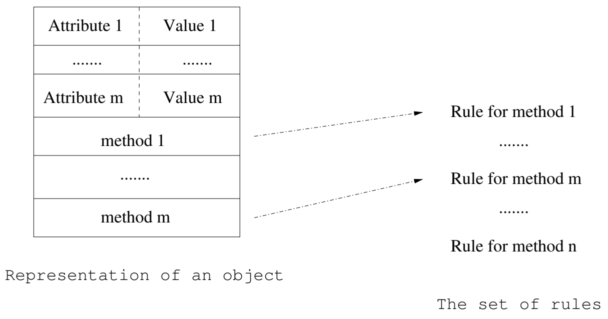


Fig. 1. Object representation

The notation used is the following: each operator takes arguments that are denoted by _. The sort of these arguments are given in the left part of the profile while the right part is the sort of the result. An operator is mixfix, that is to say that an argument can appear anywhere in the operator definition. For instance, `_(_)` has two arguments of respective sorts $MName$ and *MBody* and gives a result of sort $Method$. This is a mixfix operators that builds the association *(attribute,value)*: the name of the attribute (of sort $MName$) appears in the first place and the corresponding value (of sort *MBody*) between the brackets.

`[_]` is the constructor of objects. Each object o has the form $[LM]$ where LM, of sort *Methods* is a set composed of pairs *attribute(value)* and of references to methods. This set is built with the operator `_,_` which is associative and commutative; this is indicated by annotation (AC). Each element, of sort $Method$, is thus either a pair *(attribute,value)*, or a reference to a method (the last _ operator) also of sort $MName$.

To these declarations and definitions, we also add rules that correspond to the manipulation of objects: to add an attribute, to modify its value, to access a method and to create a new object. These rules are defined in Section 3.3. But, before defining the rules, we have to present the corresponding operators:

$$\begin{array}{ll} add(\_,\_) & : Object \times Method \mapsto Object \\ kill(\_,\_) & : Object \times MName \mapsto Object \\ access(\_,\_) & : Object \times MName \mapsto MBody \\ new(\_) & : Object \qquad\qquad\quad \mapsto Object \end{array}$$

These four basic operators are defined and used to construct and decompose objects: the `add(_,_)` operator is used to add a new element to the set of attributes and methods; the `kill(_,_)` operator is used to remove an element given in the parameters from the set of pairs and references that compose the object; the `access(_,_)` operator is used to have access to the value associated to an attribute given as parameter; the last `new` operator is used to create a new object from the object representing the class.

In the set of methods, the two particular methods `Get` and `Set` take as arguments the attribute they deal with.

$$\begin{array}{lll} Get(\_,\_) & : Object \times MName & \mapsto MBody \\ Set(\_,\_,\_) & : Object \times MName \times MBody & \mapsto Object \end{array}$$

3.3 Rules associated to the representation

We now define a system $\mathcal{R}$ of rewriting rules that are used to evaluate objects. This system is based on the above representation of objects and inspired from the object ρ-calculus [CKL01]. The whole rewrite system $\mathcal{R}$ can be found in Figure 2. $\mathcal{R}$ is composed of rules related to the object definition and manipulation. We have proved in [Dub01] that the system $\mathcal{R}$ is confluent and terminating.

Add a component	$add([LM], me) \rightarrow_{\mathcal{R}} [LM, me]$
Remove a component-1	$kill([M(B), LM], M) \rightarrow_{\mathcal{R}} [LM]$
Remove a component-2	$kill([M, LM], M) \rightarrow_{\mathcal{R}} [LM]$
Access to an attribute value	$access([M(B), LM], M) \rightarrow_{\mathcal{R}} B$
Access to a value by `Get`	$Get(o, M) \rightarrow_{\mathcal{R}} access(o, M)$
Modification of a value by `Set`	$Set(o, M, B) \rightarrow_{\mathcal{R}} add(kill(o, M), M(B))$
Creation of a new object	$new(o) \rightarrow_{\mathcal{R}}$ $[a_1(vi_1), \ldots, a_n(vi_n), m_1, \ldots, m_m]$
The operator `Geta`	$Get_a(o) \rightarrow_{\mathcal{R}} Get(o, a)$
The operator `Seta`	$Set_a(o, v) \rightarrow_{\mathcal{R}} Set(o, a, v)$
Method call for `Geta`	$[LM, Get_a].Get_a \rightarrow_{\mathcal{R}} Get_a([LM, Get_a])$
Method call for `Seta`	$[LM, Set_a].Set_a(v) \rightarrow_{\mathcal{R}} Set_a([LM, Set_a], v)$
Method call for `new`	$[LM, new].new \rightarrow_{\mathcal{R}} new([LM, new])$
Method call for a method `m`	$[LM, m].m(p_1, \ldots, p_m) \rightarrow_{\mathcal{R}}$ $m([LM, m], p_1, \ldots, p_m)$

Fig. 2. The system $\mathcal{R}$

3.4 Operators and rules associated to user's methods

For each user methods, we associate:

– a constant of sort *MName* representing the name of the method;
– an operator declaration which profile is based on the profile of the user's method: to each parameter of the method corresponds an argument of the operator. We also add to the corresponding operator a first argument which represents the object itself (the **self**);
– a set of rules defining this operator.

Let us now present how the rules associated to user's methods are built. This is performed by defining a transformation mechanism that deduces rewrite rules for each user method. The operator $build - rule(\_)$ (cf. Figure 3) takes as argument the definition of a method in the formalism described in Section 2.2 and returns the associated rewrite rule in the ELAN syntax.

```
⎧ build − rule( method name (args) for t vars body) =
⎪                             rules for t
⎪                               S : class_name;
⎪                               var − decl(vars, args, body)
⎪                               [] name(S, get − args(args)) => build − rhs(body, list_vars) end
⎪                             end
⎪ build − rule( method name (args) for t body)      =
⎪                             rules for t
⎪                               S : class_name;
⎪                               var − decl(args, body)
⎪                               [] name(S, get − args(args)) => build − rhs(body, list_vars) end
⎪                             end
⎨ build − rule( method name for t vars body)        =
⎪                             rules for t
⎪                               S : class_name;
⎪                               var − decl(vars, body)
⎪                               [] name(S) => build − rhs(body, list_vars) end
⎪                             end
⎪ build − rule( method name for t body)             =
⎪                             rules for t
⎪                               S : class_name;
⎪                               var − decl(body)
⎪                               [] name(S) => build − rhs(body, list_vars) end
⎩                             end
```

Fig. 3. Definition of function *build − rule*

In the definition of $build - rule$, we handle the four possibilities that can appear and which depend on having or not arguments ($args$) and local variable declarations ($vars$). The rules are built in the ELAN syntax.

The operator $build - rhs(\_, \_)$ takes a body and a list of variables which is built step by step. Each new variable added to this list corresponds to a local variable used when an object is modified. This list memorizes this information. When initialized, this variable list is only composed of the variable S.

Example 5 *Let us consider the Translate method presented in Exemple 2. The rule corresponding to this method is defined by build − rule and the result is:*

```
rules for PointTranslation
 N  : int;
 S  : PointTranslation;
 O1 : PointTranslation;
[] Translate(S) => O1
      if GetX(S) > GetY(S)
      where N := () GetX(S) - GetY(S)
      where O1 := () TranslateX(S,N)
```

More details can be found in [Dub01].

The system $\mathcal{R}$ presented in Section 3.3 is thus enriched with the rules defining the user methods to give a set $\mathcal{R}'$. For the time being, the system does not check that $\mathcal{R}'$ is confluent and terminating and this is under the user's responsibility. Indeed, we aimed at a complete proof environment that would automatically check these properties.

4 An operational semantics based on the ρ-calculus

Our goal is now to give a formal semantics to this object-oriented extension of ELAN. Several calculus as the *Object Lambda Calculus* defined by K. Fisher, F. Honsell and J.C. Mitchell [FHM94] and the *Object Calculus* of M. Abadi and L. Cardelli [AC96] are candidates to provide object languages with a formal semantics. Our choice is to base the semantics on another calculus called the Rewriting Calculus, or ρ-calculus defined by H. Cirstea and C. Kirchner [CK99] that encompasses in particular both λ-calculus and term rewriting. In this calculus, terms, rewriting rules and application of a rule on a term can be represented.

This choice was done for two reasons: first, the ρ-calculus already provides an operational semantics for ELAN and second, it is general enough to represent both the *Object Lambda Calculus* and the *Object Calculus* as shown in [CKL01], where an extension of the ρ-calculus, called the object ρ-calculus has been defined.

In order to prove that the ρ-calculus gives an operational semantics to the object extension of ELAN, we establish a close correspondence between reduction of an object term in the algebraic theory of objects given in Section 3 and reduction of the corresponding ρ-term in the ρ-calculus.

In the algebraic theory of objects, an object is represented as a term of sort `Object`. We have proved that the set of rules $\mathcal{R}$ is terminating and confluent and we can consider a confluent and terminating extended set $\mathcal{R}'$ with rewriting rules corresponding to user-defined methods. So each term t of sort `Object` has a normal form denoted by $NF_{\mathcal{R}'}(t)$, or $NF(t)$ for short.

In Figure 4, we illustrate that each reduction (noted $\dashrightarrow$) of a term t of sort `Object` to its normal form $NF(t)$ corresponds to a reduction (noted $\rightarrow$) in the ρ-calculus from a ρ-term t_0 to another one t_0', where t_0 and t_0' are the respective translations of t and $NF(t)$ considering a translation τ. Definition of τ, results and proofs can be found in [Dub01].

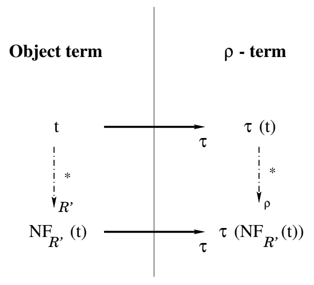


Fig. 4. Correspondence between object rewriting and ρ-term evaluation

Once object terms have been translated to ρ-terms, a data base of objects is translated to a ρ-term built with an AC-operator on top (a set of ρ-terms). Rules on objects also correspond to ρ-terms [CK99] and application of these rules to the data base is application of ρ-calculus.

5 Conclusion

The purpose of this paper was to show that objects, rewriting rules and strategies can be integrated in a same formalism with an operational semantics defined by the ρ-calculus.

We have proposed a language to define objects in the "programming by rewrite rules and strategies" paradigm offered by the ELAN system. This consists in adding OModules to the standard modules where operators, rules and strategies are defined in ELAN; in OModules, classes with attributes and methods are declared. Mixing OModules with standard modules allows the developer to define applications where special rules that manage objects can be defined. Furthermore, we have shown that objects can also be represented in the ρ-calculus that already gives an operational semantics to ELAN. The purpose of this work was to provide object features in ELAN in a semantically coherent way, without pretending to design a new powerful object oriented language. On the other hand, thanks to the operational semantics based on the rewriting calculus, one can develop verification tools that often lack in classical object oriented languages.

Developing objects in a rewriting context is also useful for rewriting itself. Indeed, when considering only a part of the object base in an object rule, this allows a better structure of the knowledge and this also allows a kind of global variable represented by the object base. This is very useful in a rewriting context where the possibility to use global variables often lacks.

In [DK00b], a formalism where rules may also be extended with constraints is presented. This leads to a general framework where rewriting rules and strategies can manage simultaneously an object base and a constraint base. As the constraint solver used in the applications definition is also based on the rewrite system ELAN, and, thus, on the ρ-calculus semantics, this complete framework is also based on the ρ-calculus semantics [Dub01].

Some applications of ELAN extended with objects have been developed. A first one consists in defining a multi-elevator controller [DK00a]. A second one consists in defining a print controller [DK00b]; in this case, rules are extended with objects and also with constraints. Planification and scheduling problems are then very easily defined in this formalism.

In such applications, strategies to control the applications of rules on objects have been proved useful. A further interesting direction is to use strategies to define methods, especially non deterministic ones, inside OModules. Although this is not a problem at the semantical level, this would need to implement in ELAN an explicit application operator.

Adding new components to the standard ELAN system such that OModules and new kind of rewrite rules was made possible by using a transformation process of OModules into ELAN modules, which relies on the algebraic theory of objects presented in this paper. The extended language has been prototyped in ELAN in this way, and more details can be found in [DK00a, Dub01]. Although this first experiment was a good approach to explore the power of the framework, in order to get an efficient programming language, one needs to go further. A more promising approach, currently explored, is to translate object programs into an internal term structure directly executable by the ELAN compiler, avoiding in this way to produce new ELAN modules.

A An example

In order to illustrate now the use of objects in the extended language, let us consider a program whose purpose is to automate and control an elevator system for buildings with multiple elevators.

Several implementations of a multi-elevator controller exist in the literature, for instance based on constraint nets [ZM93], on temporal logic [Bar85] or on the *Abstract State Machine* [Abr96]. Although not original, this example nicely illustrates the use of ours formalism based on rules, objects and strategies. Compared to other approaches, ours is uniform, the rules are clear and the ability for the user to develop easily strategies to express control is very appealing. Moreover, verification techniques available in rule formalism can be adapted to prove properties such as termination, confluence, completeness of the specification.

To formalize the multi-elevator controller, we first define two classes: a class `MLift` for elevators and a class `Call` for the controller. Then, we present the rules defined to design the multi-elevator controller and, finally, the strategies before a short presentation of an execution.

A.1 The class `MLift`

This class describes elevators. Each elevator is an object of this class, characterized by:

– its current floor denoted by the attribute `CF` -*current floor*- represented by an integer,
– its state: is it going up?, down?, or is it waiting for a call? This is defined by the attribute `State` of sort `LiftState`,
– its list of floors where it has to stop with the attribute `LStop` which is a list of integers.

The sort describing the state of an elevator is called `LiftState`. This sort is defined with two operators: a constant `Wait` of sort `LiftState` and an operator `Move(_)` which takes a term of sort `Direction` as argument (`Up` and `Down` are of sort `Direction`) and returns an element of sort `LiftState`. This is defined in a standard ELAN module by:

```
operators global
    Up        : Sense;
    Down      : Sense;
    Move(@)   : (Sense) LiftState;
    Wait      : LiftState;
end
```

Three other attributes are also defined:

– `Zone` that indicates the zone where this elevator is. Indeed, each building is divided in several zones, each one composed by consecutive floors, and there is as many zones as elevators. For instance, considering a building with 4 elevators and with 27 floors, four zones will be considered: the first one from floor 0 to 6, then, a second one (floor 7 to 13), a third one (floor 14 to 20) and a last fourth one form floor 21 to 27. This attribute is useful if we want to guarantee that when dividing the number of floors into a number of zones, each zone does not have more than two elevators working at any moment. This guarantees a better quality of service for the access to an elevator in the building.
– `F` (standing for Flag) whose value is either 0 or 1 indicates that an elevator is performing an instruction (`F` to 1) or waiting for a new instruction (`F` to 0).
– `I`, standing for Interruption, is an integer which can take value in $\{0, 1\}$. If `I`$= 0$, then the elevator has no interruption and is available; if `I`$= 1$, then, the elevator is out of service.

Several methods are defined for the class `MLift`:

– a method `WhichSense(_)` takes an integer representing the new floor that the elevator has to reach from its current one. It returns the direction that the elevator has to adopt: either it will go up or down.
– the method `UpdateZone` computes, after the elevator has moved, in which zone it is.
– a method `AddLStop(_)` takes a list of integers `L` representing a list of different floors and returns the object representing the elevator whose attribute `LStop` (list of floors) has been updated with `L`. The new list `LStop` is sorted.

– the last method, RemoveLStop(_), deletes from the list of stops LStop the floor given as parameter of this method.

These definitions and declarations of methods and attributes are grouped together in the definition of the OModule for the class MLift:

```
class MLift

imports ToolsMLift

attributes CF:int = 0
           State:LiftState = Wait
           LStop:list[int] = nil
           Zone:int = 0
           F:int = 0
           I:int = 0

method WhichSense(N:int) for MLift
 S : Sense;
 <S:=ChooseSense(self.GetCF,N) ; self.SetState(Move(S))>

method UpdateZone for MLift
 <self.SetZone(NewZone(self.GetCF))>

method AddLStop(L:list[int]) for MLift
 <self.SetLStop(AddAndSort(self.GetLStop,L))>

method RemoveLStop(N:int) for MLift
 <self.SetLStop(RemoveList(self.GetLStop,N))>
End
```

A.2 The class Call

This class describes the central memory for the multi-elevator controller. When people enter the elevator, they select floors where they want to go out. This is formalized by an attribute LCall composed of a list of integers: the requested floor. These are floors that have to be served to load people.

To distinguish calls that are processed from those that are waiting, a second attribute AssignedCall is composed of calls that have been assigned to an elevator and which are to be processed.

Different methods are defined in the object module describing the class Call:

– a method AddAssignedCall(_) takes an integer that represents a floor and adds it in the list of calls that are processed.
– a method RemoveAssignedCall(_) takes an integer that represents a floor and removes it from the list of calls that are currently processed.
– a last method called RemoveLCall(_) takes an integer that represents a floor and removes it from the list of calls that are waiting to be processed.

The class `Call` is defined in the following OModule:

```
class Call

imports Tools

attributes LCall:list[int] = nil
           AssignedCall:list[int] = nil

method AddAssignedCall(N:int) for Call
 <self.SetAssignedCall(AddList(self.GetAssignedCall,N))>

method RemoveAssignedCall(N:int) for Call
 <self.SetAssignedCall(RemoveList(self.GetAssignedCall,N))>

method RemoveLCall(N:int) for Call
 <self.SetLCall(RemoveList(self.GetLCall,N))>

End
```

A.3 The rules

The rules that define the actions on elevators can now be described.

The two main rules are the rule `Up` and the rule `Down`. An elevator going upward or downward can continue if the current floor is not a floor occurring in its list of stops or in the list of calls. If the elevator can continue, the current floor and the zone are updated. A condition to apply these rules is that the value of the flag is 0; this value is updated to 1 after application.

```
[Up] O1:MLift::[State(Move(Up)) , F(0) , I(0)]
     O2:LCall
        =>
      O1(CF<-O1.CF+1).UpdateZone(F<-1)
      O2
         if not(in(O1.CF,O1.Stop))
         if not(in(O1.CF,O2.LCall))

[Down] O1:MLift::[State(Move(Down)) , F(0) , I(0)]
       O2:LCall
         =>
        O1(CF<-O1.CF-1).UpdateZone(F<-1)
        O2
         if not(in(O1.CF,O1.Stop))
         if not(in(O1.CF,O2.LCall))
```

Each elevator can change its moving direction in two cases: either it has reached the top level (or the bottom level), or its current floor is greater (resp. lower) than the maximum (resp. the minimum) level where it has to stop. This is represented by these two rules `ChangeToDown` and `ChangeToUp`:

```
[ChangeToDown]
    O1:MLift::[State(Move(Up)) , F(0) , I(0)]
      =>
    O1(State<-Move(Down),F<-1)
      if O1.CF > Max(O1.LStop) or O1.CF == MaxLevel

[ChangeToUp]
    O1:MLift::[State(Move(Down)) , F(0) , I(0)]
      =>
    O1(State<-Move(Up),F<-1)
      if O1.CF < Min(O1.LStop) or O1.CF == MinLevel
```

Each elevator has to stop for different reasons. An elevator stops when its current floor is in its list of requested stops (rule `OpenDoorsStop`) or when it is in the list of calls (rule `OpenDoorsCall`). These rules can be applied in the two moving directions; this corresponds to the variable `S` for the attribute `State`.

If the rule `OpenDoorsStop` is applied, the current floor is removed from the list of stops. If the rule `OpenDoorsCall` is applied, the current floor is also removed from the list of calls and then, the new stops requested by people entering the elevator are added to the list of stops.

```
[OpenDoorsStop]
    O1:MLift::[F(0) , I(0)]
        =>
    O1.RemoveLStop(O1.CF)(F<-1)
        if O1.State != Wait
        if in(O1.CF,O1.LStop)

[OpenDoorsCall]
    O1:MLift::[F(0) , I(0)]
    O2:Call
       =>
    O1.AddLStop(L1)(F<-1)
    O2.RemoveLCall(O1.CF)
       if O1.State != Wait
       if in(O1.CF,O2.LCall)
       where L1 := () ObtainNewStops(O1.CF)
```

If the current floor of an elevator is in the list of calls and in the list of stops, instead of applying consecutively the two previous rules, we just apply one rule labelled `OpenDoorsStopAndCall`.

```
[OpenDoorsStopAndCall]
     O1:MLift::[F(0) , I(0)]
     O2:Call
       =>
     O1.AddLStop(L1).RemoveLStop(O1.CF)(F<-1)
     O2.RemoveLCall(O1.CF)
       if O1.State != Wait
```

```
        if in(O1.CF,O1.LStop)
        if in(O1.CF,O2.LCall)
        where L1 := () ObtainNewStops(O1.CF)
```

A feature of this multi-elevator controller is that priority is given to a call, and once it has been served, other requested stops are served.

A call is assigned to an elevator whose `State` value is `Wait`. This is done by the rule `AssignACall`. When an elevator can be selected (i.e. there is at least a floor calling an elevator), we compute which floor is selected (this is the nearest one and we call it `NextFloor`) by the function `ChooseNextFloor`. Then, the two objects are updated by removing `NextFloor` from the list of calls, by adding it to the list of assigned calls in the central memory and to the list of stops, and by choosing the good direction to go for the selected elevator.

```
[AssignACall]
    O1:MLift::[State(Wait) , F(0) , I(0)]
    O2:Call
       =>
    O1.WhichSense(NextFloor).AddLStop(NextFloor.nil)(F<-1)
    O2.AddAssignedCall(NextFloor).RemoveLCall(NextFloor)
     if O2.LCall != nil
     where NextFloor := () ChooseNextFloor(O1.CF,O2.LCall)
```

When the elevator reaches a floor, we test if this floor is assigned to it, we apply the rule `OpenDoorsAssignedCall`, that updates the list of stops and the list of assigned calls. It also uses the function `ObtainNewStops` that asks people inside the elevator which floors they want to go to.

```
[OpenDoorsAssignedCall]
    O1:MLift::[F(0) , I(0)]
    O2:Call
       =>
    O1.AddLStop(L1).RemoveLStop(O1.CF)(F<-1)
    O2.RemoveAssignedCall(O1.CF)
       if O1.State != Wait
       if in(O1.CF,O2.AssignedCall)
       where L1 := () ObtainNewStops(O1.CF)
```

A condition to assign a call to an elevator is that at least one elevator has the attribute `State` to `Wait`. This is possible only when its list of stops is empty as shown in the rule `Wait`:

```
[Wait]  O1:MLift::[F(0) , I(0)]
          =>
        O1(State<-Wait)
          if O1.State != Wait
          if O1.LStop = nil
```

A.4 Strategies

In general, the application of a rule on the data base of objects may return several results, for instance when several objects or multisets of objects match its left-hand side. This introduces some non-determinism and the need to control rule application. This is why the concept of strategy is introduced. Strategies are used to control the application of rules on objects: strategies provide the capability to define a sequential application of rules; to make choices between several rules that can be applied; to iterate rules; etc.

For the previous example, we define a few strategies to guide the application of the rules on the data base of objects.

The first one is `ONELIFT` which tries to assign a call to a waiting lift; then, it tries to open the doors of the elevator at current floor if, 1- the floor corresponds to an assigned call, 2- it corresponds to a stop and a call, 3- it corresponds only to a call or 4- only to a stop. If the current floor is not a floor where a stop is required, it checks if the direction of the elevator has to be changed and, otherwise, it continues to go upward or downward.

```
[] ONELIFT   => first(  AssignACall ,
                        OpenDoorsAssignedall ,
                        OpenDoorsStopAndCall ,
                        OpenDoorsCall ,
                        OpenDoorsStop ,
                        ChangeSenseToDown ,
                        ChangeSenseToUp ,
                        Up ,
                        Down)
end
```

This strategy is applied as long as there is an elevator whose flag is not set at 1. To work on a set of elevators, we define the strategy `ALLLIFTS`:

```
[] ALLLIFTS      => repeat*(Wait) ;
                    repeat*(ONELIFT) ;
                    repeat*(RemoveFlag)
end
```

The rule `RemoveFlag` removes all flags at 1 and put them at 0. To go from an initial situation to a situation where all floors are served and where nobody is waiting inside an elevator, we define a main strategy `MAIN` that repeats the rule `Main` until the data base of elevators does not change.

```
[] MAIN => first one (repeat*(Main))
end
```

The labelled rule `Main` is defined as:

```
[Main] ST => ST1
          where ST1 := (ALLLIFTS) ST
          if ST1 != ST
```

A.5 The execution

Let us consider an initial situation described as:

```
O(1):MLift::[CF(14), State(Wait), Zone(1), LStop(nil), F(0), I(0)]
O(2):MLift::[CF(11), State(Wait), Zone(1), LStop(nil), F(0), I(0)]
O(3):MLift::[CF(2) , State(Wait), Zone(0), LStop(nil), F(0), I(0)]
O(4):Call::[AssignedCall(nil), LCall(3.9.17.24.nil)]
```

Let us assume that the ground floor is floor 0 and the top level is the level 25. In this initial situation, we have three lifts `O(1)`, `O(2)` and `O(3)`. The first one is waiting at floor 2, the second at floor 11 and the last one at level 14. Four levels are calling an elevator: the 3rd, 9th, 17th and 24th ones.

Applying the `MAIN` strategy to this initial term leads to the following execution:

```
O(1):MLift::[CF(14),State(Move(Up))  ,Zone(1),LStop(17.nil),F(0),I(0)]
O(2):MLift::[CF(11),State(Move(Down)),Zone(1),LStop(9.nil) ,F(0),I(0)]
O(3):MLift::[CF(2) ,State(Move(Up))  ,Zone(0),LStop(3.nil) ,F(0),I(0)]
O(4):Call::[AssignedCall(3.9.17.nil) , LCall(24.nil)]

O(1):MLift::[CF(15),State(Move(Up))  ,Zone(1),LStop(17.nil),F(0),I(0)]
O(2):MLift::[CF(10),State(Move(Down)),Zone(1),LStop(9.nil) ,F(0),I(0)]
O(3):MLift::[CF(3) ,State(Move(Up))  ,Zone(0),LStop(3.nil) ,F(0),I(0)]
O(4):Call::[AssignedCall(3.9.17.nil) , LCall(24.nil)]

An elevator is stopped at level 3, please enter the desired stops
as a list of sorted integers separated by . and terminated by end:
```

After the user has entered different stops that will be considered by the elevator controller, several execution steps occur and lead, finally, to the two last steps below:

```
...
  O(1):MLift::[CF(17), State(Wait)    , Zone(1), LStop(nil), F(0), I(0)]
  O(2):MLift::[CF(10), State(Wait)    , Zone(1), LStop(nil), F(0), I(0)]
  O(3):MLift::[CF(24), State(Move(Up)), Zone(2), LStop(nil), F(0), I(0)]
  O(4):Call::[AssignedCall(nil), LCall(nil)]

  O(1):MLift::[CF(24), State(Wait), Zone(2), LStop(nil), F(0), I(0)]
  O(2):MLift::[CF(17), State(Wait), Zone(1), LStop(nil), F(0), I(0)]
  O(3):MLift::[CF(10), State(Wait), Zone(1), LStop(nil), F(0), I(0)]
  O(4):Call::[AssignedCall(nil), LCall(nil)]
```

During this execution, we observe the evolution of the set of elevators step by step:

1. At 1st step, three calls are assigned (these three calls are put in the attribute `AssignedCall` of object `O(4)`), one to elevator `O(1)` (the 17th floor), one to

the elevator `O(2)` (the 9th floor) and one to the elevator `O(3)` (the 3rd floor). One call has not yet been assigned. This assignment step of calls also selects a direction for each elevator (two go up and one down).
2. The 2nd step does not change a lot of attributes. Each elevator goes on up or down. We can notice that one elevator, the one called `O(3)`, has reached the requested 3rd floor. Then, the operation `ObtainNewStops` can be performed and it consists in asking user new stops for this elevator. This uses the inputs/outputs of the ELAN system.
3. This process continues for a few steps...
4. The last but one step has no more call. Two elevators are waiting (`O(1)` and `O(2)`) and the `O(3)` elevator is going down, it is at floor 24 without any floor to serve.
5. The last step makes the previous elevator waiting at floor 24. This step cannot be reduced anymore, this is the result term.

References

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer-Verlag, 1996.

[ADH+98] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams IV, D. P. Friedman, E. Kohlbecker, G. L. Steele Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, August 1998.

[AG96] K. Arnold and J. Gosling. *The Java programming language.* Addison Wesley, 1996.

[Bar85] H. Barringer. Up and down the temporal way. Technical Report UMCS–85–9–3, Department of Computer Science, Manchester University, Oxford Rd., Manchester M13 9PL, UK, 1985.

[BCD+00] P. Borovanský, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen, and M. Vittek. ELAN *V 3.4 User Manual.* LORIA, Nancy (France), fourth edition, January 2000.

[BKKR01] P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: a functional semantics. *International Journal of Foundations of Computer Science*, 2001.

[CDE+00] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martì-Oliet, J. Meseguer, and J.F. Quesada. Towards Maude 2.0. In K. Futatsugi, editor, *WRLA2000, the 3rd International Workshop on Rewriting Logic and its Applications, September 2000, Kanazawa, Japon.* Electronic Notes in Theoretical Computer Science, 2000.

[Cir00] H. Cirstea. *Le Rho Calcul: Fondements et Applications.* PhD thesis, Université Henri Poincaré - Nancy 1, 2000.

[CK99] H. Cirstea and C. Kirchner. Combining higher-order and first-order computation using ρ-calculus: Towards a semantics of ELAN. In D. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, Research Studies, ISBN 0863802524, pages 95–120. Wiley, 1999.

[CKL01] H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. In A. Middeldorp, editor, *Proceedings 12th Conference on Rewriting Techniques and Applications, RTA 2001, Utrecht, The Netherlands*, volume 2051 of *Lecture Notes in Computer Science*, pages 77–92. Springer-Verlag, 2001.

[CL96] Y. Caseau and F. Laburthe. CLAIRE: Combining objects and rules for problem solving. In *Proceedings of the JICSLP'96 workshop on multi-paradigm logic programming, TU Berlin, Germany*, 1996.

[DK00a] H. Dubois and H. Kirchner. Objects, rules and strategies in ELAN. In *Proceedings of the second AMAST workshop on Algebraic Methods in Language Processing, Iowa City, Iowa, USA*, May 2000.

[DK00b] H. Dubois and H. Kirchner. Rule Based Programming with Constraints and Strategies. In K.R. Apt, A.C. A. C. Kakas, E. Monfroy, and F. Rossi, editors, *New Trends in Constraints, Papers from the Joint ERCIM/Compulog-Net Workshop, Cyprus, October 25-27, 1999*, volume 1865 of *Lecture Notes in Artificial Intelligence*, pages 274–297. Springer-Verlag, 2000.

[Dub01] H. Dubois. *Systèmes de Règles de Production et Calcul de Réécriture*. PhD thesis, Université Henri Poincaré - Nancy 1, 2001.

[FHM94] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specializatio n. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[HSW95] M. Henz, G. Smolka, and J. Wurtz. Object-oriented concurrent constraint programming in oz. In V. Saraswat and P. van Hentenryck, editors, *Principles and Practice of Constraint Programming.*, pages 27–48. MIT Press, 1995.

[Mes89] J. Meseguer. General logics. In H.-D. Ebbinghaus et al., editor, *Logic Colloquium'87*, pages 275–329. Elsevier Science Publishers B. V. (North-Holland), 1989.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mey92] B. Meyer. *Eiffel: The Language.* Prentice Hall, 1992.

[Ros92] J.-P. Rosen. What Orientation Should Ada Objects Take? *Communications of the ACM*, 35(11):71–76, 1992.

[RV98] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension of ML. *Theory and Practice of Object Systems*, 4(1):27–52, 1998.

[Ste90] G. Steele, Jr. *Common* LISP: *The Language.* Digital Press, Bedford, Massachusetts, 2nd edition, 1990.

[ZM93] Y. Zhang and Alan K. Mackworth. Design and analysis of embedded real-time systems: An elevator case study. Technical Report TR-93-04, Department of Computer Science, University of British Columbia, February 1993.

Modelica - A Declarative Object Oriented Multi-Paradigm Language.

Peter Fritzson, Peter Bunus

Department of Computer and Information Science, Linköping University,
SE 581-32, Linköping, Sweden
{petfr,petbu}@ida.liu.se

Abstract. Modelica is a general multi-paradigm object-oriented language for continuous and discrete-event specification of complex systems for the purpose of efficient simulation, computational applications, architecture system specification, etc. The Modelica modeling language and technology is being warmly received by the world community in modeling and simulation. It is bringing about a revolution in this area, based on its ease of use, visual design of models with combination of Lego-like predefined model building blocks, its ability to define model libraries with re-usable components and its support for specifying complex systems involving parts from several application domains. In this paper we present the Modelica language with emphasis on its multi-paradigm language features including functional, object-oriented, constraint-based, architectural, concurrent, and visual programming.

1 Introduction

Modelica is a new multi-paradigm language for hierarchical object-oriented modeling and computational applications, which is developed through an international effort [3][6][2]. The language is one of the rare examples of a programming language that combines declarative functional programming with object-oriented programming. In fact, Modelica integrates several programming paradigms, which will be illustrated throughout the paper:

– Declarative functional programming using equations and functions without side effects.
– Object-oriented programming.
– Constraint programming based on equations.
– Architectural system specification with connectors and components.
– Concurrency and discrete event programming, based on the synchronous data flow principle.
– Visual programming based on connecting icons with ports, and hierarchical decomposition.

Additionally, the multi-domain capability of Modelica gives the user the possibility to combine model components from different application domains within the same application model, e.g. combining electrical, mechanical, hydraulic, thermodynamic, control, algorithmic components. Modelica is primarily a modeling language, sometimes called hardware description language, that allows the user to specify mathematical models of complex systems, e.g. for the purpose of computer simulation of dynamic systems where behavior evolves as a function of time. Modelica is also a declarative object-oriented equation based programming language, oriented towards computational applications with high complexity requiring high performance.

The execution performance of algorithmic Modelica code is close to comparable code in the C language, whereas the simulation performance for equation-based models often is better than hand programmed models in conventional programming languages due to the advanced symbolic optimization performed by the Modelica compiler.

The four most important features of Modelica are:

– Modelica is primarily based on equations instead of assignment statements. This permits acausal modeling that gives better reuse of classes since equations do not specify a certain data flow direction. Thus a Modelica class can adapt to more than one data flow context. Algorithmic constructs including assignments are also available in a way that does not destroy the declarative properties of the language.
– Modelica has multi-domain modeling capability, meaning that model components corresponding to physical objects from several different domains such as e.g. electrical, mechanical, thermodynamic, hydraulic, biological and control applications can be described and connected.
– Modelica is an object-oriented language with a general class concept that unifies classes, generics - known as templates in C++, and general subtyping into a single language construct. This facilitates reuse of components and evolution of models. The class concept is used consistently throughout the language, e.g. packages as well as primitive types such as Integer and Real are classes with no loss of performance.
– Modelica has a strong software component model, with constructs for creating and connecting components. Thus the language is ideally suited as an architectural description language for complex systems.

The reader of the paper is referred to [7][8] and [4][9] for a complete description of the language and its functionality from the perspective of the motivations and design goals of the researchers who developed it.

2 Object-Oriented Mathematical Modeling

Traditional object-oriented programming languages like Simula, C++, Java, and Smalltalk, as well as procedural languages such as Fortran or C, support programming with operations on stored data. The stored data of the program includes variable values and object data. The number of objects often changes dynamically. The Smalltalk view of object-orientation emphasizes sending messages between (dynamically) created objects.

The Modelica view on object-orientation is different since the Modelica language emphasizes *structured* mathematical modeling. Object-orientation is viewed as a structuring concept that is used to handle the complexity of large system descriptions. A Modelica model is primarily a declarative mathematical description, which simplifies further analysis. Dynamic system properties are expressed in a declarative way through equations.

The concept of *declarative* programming is inspired by mathematics where it is common to state or declare what holds, rather than giving a detailed stepwise algorithm on how to achieve the desired goal as is required when using procedural languages. This relieves the programmer from the burden of keeping track of such details. Furthermore, the code becomes more concise and easier to change without introducing errors.

Thus, the Modelica view of object-orientation, from the point of view of object-oriented mathematical modeling, can be summarized as follows:

– Object-orientation is primarily used as a structuring concept, emphasizing the declarative structure and reuse of mathematical models.
– Dynamic model properties are expressed in a declarative way through equations.
– An object is a collection of instance variables and equations that share a set of stored data.
– Object orientation is not viewed as dynamic message passing.

The declarative object-oriented way of describing systems and their behavior offered by Modelica is at a higher level of abstraction than the usual object-oriented programming since some implementation details can be omitted. For example, the users do not need to write code to explicitly transport data between objects through assignment statements or message passing code. Such code is generated automatically by the Modelica compiler based on the given equations.

Just as in ordinary object-oriented languages, classes are blueprints for creating objects. Both variables and equations can be inherited between classes. Function definitions can also be inherited. However, specifying behavior is primarily done through equations instead of via methods. There are also facilities for stating algorithmic code including functions in Modelica, but this is an exception rather than the rule.

As briefly mentioned before, acausal modeling is a declarative modeling style meaning modeling based on equations instead of assignment statements. The main advantage is that the solution direction of equations will adapt to the data flow context in which the solution is computed. The data flow context is defined by stating which variables are needed as *outputs*, and which are external inputs to the simulated system. The acausality of Modelica library classes makes these more reusable than traditional classes containing assignment statements where the *input-output* causality is fixed.

To illustrate the idea of acausal physical modeling we give an example of a simple electrical circuit, see Fig 1. The connection diagram of the electrical circuit shows how the components are connected and roughly corresponds to the physical layout of the electrical circuit on a printed circuit board. The physical connections in the real circuit correspond to the logical connections in the diagram. Therefore the term physical modeling is quite appropriate.

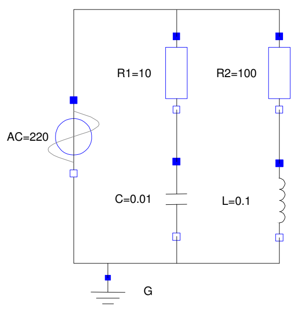


Fig. 1. Connection diagram of the acausal simple circuit model.

The Modelica `SimpleCircuit` class below directly corresponds to the circuit depicted in the connection diagram of Fig 1. Each graphic object in the diagram corresponds to a declared instance in the simple circuit model. The model is acausal since no signal flow, i.e. cause-and-effect flow, is specified. Connections between objects are specified using the `connect` statement, which is a special syntactic form of equation that we will tell more about later.

```
class SimpleCircuit
   Resistor  R1(R=10);
   Capacitor C(C=0.01);
   Resistor  R2(R=100);
   Inductor  L(L=0.1);
   Ground    G;
   VsourceAC AC;
equation
   connect (AC.p, R1.p); // Capacitor circuit
   connect (R1.n, C.p);
   connect (C.n,  AC.n);
   connect (R1.p, R2.p); // Inductor circuit
   connect (R2.n, L.p);
   connect (L.n,  C.n);
   connect (AC.n, G.p);  // Ground
end SimpleCircuit;
```

We simulate the above model during the time period `{0,0.1}`:

```
Simulate[Circuit, {t,0,0.1}];
```

Let us plot the current in the inductor for the first 0.1 second.

```
PlotSimulation[L.i[t], {t,0,0.1}];
```

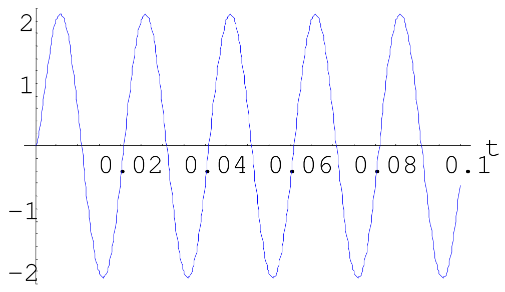


Fig. 2. Plot of the current through the inductor, `L.i`, from a simulation of the `SimpleCircuit` model.

2.1 Modelica Classes

Modelica, like any object-oriented computer language, provides the notions of classes and objects, also called instances, as a tool for solving modeling and programming problems. Every object in Modelica has a class that defines its data and behavior. A class has three kinds of members:

– *Fields* are data variables associated with a class and its instances. Fields store results of computations caused by solving the equations of a class together with equations from other classes.
– *Equations* specify the behavior of a class. The way in which the equations interact with equations from other classes determines the solution process, i.e. program execution.
– *Classes* including functions can be members of other classes.

Here is the declaration of a simple class that might represent a point in a three-dimensional space:

```
class Point "point in a three-dimensional space"
  public Real x;
  Real y, z;
end Point;
```

2.2 Inheritance

One of the major benefits of object-orientation is the ability to extend the behavior and properties of an existing class. The original class, known as the *superclass* or *base class*, is extended to create a more specialized version of that class, known as the *subclass* or *derived class*. In this process, the behavior and properties of the original class in the form of field declarations, equations, and other contents is reused, or inherited, by the subclass.

Let us regard an example of extending a simple Modelica class, e.g. the class `Point` introduced previously. First we introduce two classes named `ColorData` and `Color`, where `Color` inherits the data fields to represent the color from class `ColorData` and adds an equation as a constraint. The new class `ColoredPoint` inherits from multiple classes, i.e. uses multiple inheritance, to get the position fields from class Point and the color fields together with the equation from class `Color`.

```
record ColorData
  Real red;
  Real blue;

  Real green;
end ColorData;

class Color
  extends ColorData;
equation
  red + blue + green = 1;
end Color;

class Point
  public Real x;
  Real y, z;

end Point;

class ColoredPoint   // Multiple inheritance
  extends Point;
  extends Color;
end ColoredPoint;
```

2.3 Equations in Modelica

As we already stated, Modelica is primarily an equation-based language in contrast to ordinary programming languages where assignment statements proliferate. Equations are more flexible than assignments since they do not prescribe a certain data flow direction. This is the key to the physical modeling capabilities and increased reuse potential of Modelica classes.

Thinking in equations is a bit unusual for most programmers. In Modelica the following holds:

- Assignment statements in conventional languages are usually represented as equations in Modelica.
- Attribute assignments are represented as equations.
- Connections between objects generate equations.

Equations are more powerful than assignment statements. For example, consider a resistor equation where the resistance `R` multiplied by the current `i` is equal to the voltage `v`:

```
R*i = v;
```

This equation can be used in three ways corresponding to three possible assignment statements: computing the current from the voltage and the resistance, computing the voltage from the resistance and the current, or computing the resistance from the voltage and the current. This is expressed in the following three assignment statements:

```
i := v/R;
v := R*i;
R := v/i;
```

2.4 The Modelica Notion of Subtypes

The notion of subtyping in Modelica is influenced by a type theory of Abadi and Cardelli [1]. The notion of inheritance in Modelica is independent from the notion of subtyping. According to the definition, a class `A` is a subtype of a class `B` if and only if the class `A` contains all the public variables declared in the class `B`, and the types of these variables are subtypes of types of corresponding variables in `B`. The main benefit of this definition is additional flexibility in the definition and usage of types. For instance, the class `TempResistor` is a subtype of `Resistor`, without being a subclass of `Resistor`.

```
class Resistor "Ideal electrical resistor"
  extends TwoPin;
  parameter Real R(Unit="Ohm") "Resistance";
equation
  v = R*i;

end Resistor;

model TempResistor
  extends TwoPin;
  parameter Real R  "Resistance at reference Temp.";
  parameter Real RT=0   "Temp. dependent Resistance.";
```

```
   parameter Real Tref=20 "Reference temperature.";
   Real Temp=20      "Actual temperature";
 equation
   v = i*(R + RT*(Temp-Tref)); end TempResistor;
```

Subtyping compatibility is checked when the class is used. If a variable `a` is of type `A`, and `A` is a subtype of `B`, then the variable `a` can be initialized by a variable of type `B`.

Note that `TempResistor` does not inherit the `Resistor` class. There are different definition for evaluation of `v`. If equations are inherited from `Resistor` then the set of equations will become inconsistent in `TempResistor`, since there would be two definitions of `v`. For example, the specialized equation below from `TempResistor`:

```
v=i*(R+RT*(Temp-Tref))
```

and the general equation from class `Resistor`:

```
v=R*i
```

are incompatible. Modelica currently does not support explicitly named equations and replacement of equations, except for the cases when the equations are collected into a local class, or a declaration equation is present in a variable declaration.

2.5 Components

Components are connected via the *connection mechanism* realized by the Modelica language, which can be visualized in connection diagrams. A *component framework* realizes components and connections, and insures that communication works over the connections. For systems composed of acausal components the direction of data flow, i.e. the *causality*, is initially unspecified for connections between those components. Instead the causality is automatically deduced by the compiler when needed. Components have well-defined *interfaces* consisting of ports, also known as connectors, to the external world. These concepts are illustrated in Fig 3.
Modelica uses a slightly different terminology compared to most literature on software component systems: *connector* and *connection*, rather than *port* and *connector* respectively in software component literature.

In the context of Modelica class libraries software components are Modelica classes. However when building particular models, components are *instances* of those Modelica classes. A component class should be defined independently of the environment where it is used, which is essential for its reusability. This means that in the definition of the component including its equations, only local variables and connector variables can be used. No means of communication between a component and the rest of the system, apart from going via a connector, is then allowed. A component may internally consist of other connected components, i.e. hierarchical modeling.

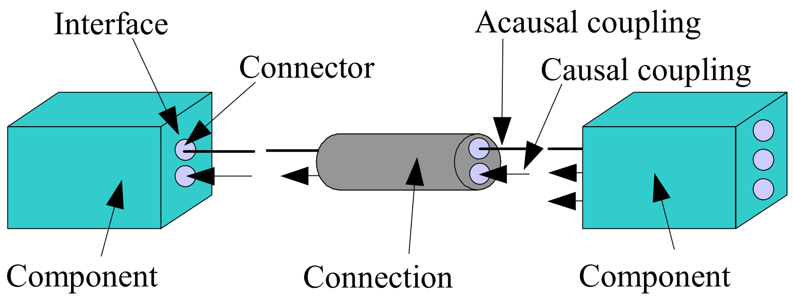


Fig. 3. Connecting two components within a component framework.

2.6 Connectors and Connector Classes

Modelica *connectors* are instances of *connector classes*, i.e. classes with the keyword `connector` or classes with the `class` keyword that fulfill the constraints of connector restricted classes. Such *connectors* declare variables that are part of the communication *interface* of a component defined by the connectors of that component. Thus, connectors specify the interface for interaction between a component and its surroundings.

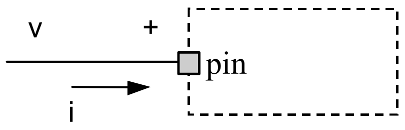


Fig. 4. A component with an electrical pin connector; i.e. an instance of a `Pin`.

For example, class `Pin` is a connector class that can be used to specify the external interface for electrical components that have pins as interaction points.

```
connector Pin
  Voltage      v;
  flow Current i;
end Pin; Pin pin; // An instance pin of class Pin
```

2.7 Connections

Connections between components can be established between connectors of equivalent type. Modelica supports equation-based acausal connections, which means that connections are realized as equations. For acausal connections, the direction of data flow in the connection need not be known. Additionally, causal connections can be established by connecting a connector with an input attribute to a connector declared as `output`.

Two types of coupling can be established by connections depending on whether the variables in the connected connectors are non-flow (default), or declared using the prefix `flow`:

- Equality coupling, for non-flow variables, according to Kirchhoff's first law.
- Sum-to-zero coupling, for flow variables, according to Kirchhoff's current law.

For example, the keyword `flow` for the variable `i` of type `Current` in the `Pin` connector class indicates that all currents in connected pins are summed to zero, according to Kirchhoff's current law.

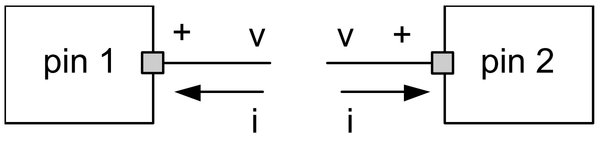


Fig. 5. Connecting two components that have electrical pins.

Connection statements are used to connect instances of connection classes. A connection statement `connect(pin1,pin2)` with `pin1` and `pin2` of connector class `Pin`, connects the two pins so that they form one node. This produces two equations, namely:

```
pin1.v = pin2.v
pin1.i + pin2.i = 0
```

The first equation says that the voltages of the connected wire ends are the same. The second equation corresponds to Kirchhoff's current law saying that the currents sum to zero at a node (assuming positive value while flowing into the component). The sum-to-zero equations are generated when the prefix `flow` is used. Similar laws apply to flows in piping networks and to forces and torques in mechanical systems.

3 Discrete Event Programming

Physical systems evolve continuously over time, whereas certain man-made systems evolve by discrete steps between states. These discrete changes, or *events*, when something happens, occur at certain points in time. We talk about *discrete event dynamic systems* as opposed to *continuous dynamic systems* directly based on the physical laws derived from first principles, e.g. from conservation of energy, matter, or momentum. One of the key issues concerning the use of events for modeling is how to express behavior associated with events. The traditional imperative way of thinking about a behavior at events is a piece of code that is activated when an event condition becomes true and then executes certain actions, e.g. as in the when-statement skeleton below:

```
when (event_conditions) then
  event-action1;
  event-action2;
  ...
end when;
```

On the other hand, a declarative view of behaviour can be based on equations. *When-equations* are used to express equations that are only valid (become active) at events, e.g. at discontinuities or when certain conditions become true. The conditional *equations* automatically contain only *discrete-time* expressions (see below) since they become active only at event instants, i.e. at discrete points in time. Discrete events in Modelica take no time and are based on the synchronous data flow principle. In fact, the semantics of events can be completely defined in terms of conditional equations.

```
when <conditions> then
  <equations>
end when;
```

For example, the two equations in the when-clause below become active at the event instant when the Boolean expression `x > 2` becomes true.

```
when x > 2 then
  y1 = sin(x);
  y3 = 2*x + y1+y2;
end when;
```

So-called *discrete-time* variables in Modelica only change value at discrete points in time, i.e. at event instants, and keep their values constant between events. This is in contrast to continuous-time variables which may change value at any time, and usually evolve continuously over time, see Fig 6.

3.1 A Simple Discrete Simulation Model

A common example of discrete time simulation is a queuing system serving customers. We start by first defining a model which generates customers at random points in time moments. This model calls the function `normalvariate` which is a random number generator function (functional, without side effects) used to generate the time delay until the next customer arrival. Here we only show the `CustomerGeneration` class of the total model.

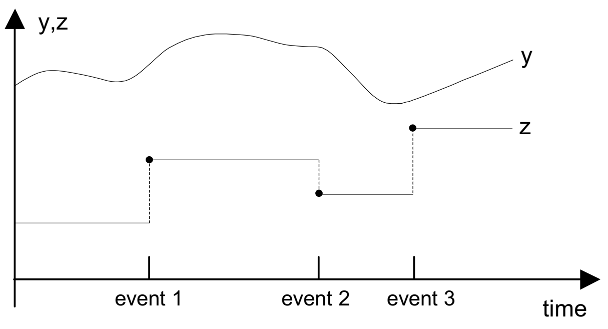


Fig. 6. A discrete-time variable z changes value only at event instants, whereas continuous-time variables like y may change value both between and at events.

```
class CustomerGeneration
  Random.discreteConnector dOutput;
  parameter Real mean = 0;
  parameter Real stDeviation = 1;
  discrete Real normalDelta;
  discrete Real nextCustomerArrivalTime(start=0);
  discrete Random.Seed randomSeed(start={23,87,187});
equation
  when pre(nextCustomerArrivalTime)<=time then
   (normalDelta,randomSeed)=
         normalvariate(mean,stDeviation,pre(randomSeed));
    nextCustomerArrivalTime = pre(nextCustomerArrivalTime)
                              + abs(normalDelta);
  end when;
  dOutput.dcon = (nextCustomerArrivalTime <>
     pre(nextCustomerArrivalTime); end CustomerGeneration;

Simulate[CustomerGeneration,{t,0,10}]
PlotSimulation[nextCustomerArrivalTime[t],{t,0,10}]
```

3.2 Game of Life Simulation

Another discrete-event example is a simple model representing Conway's game of life. Life is played on a grid of square cells where a cell can be live or dead. In our example we represent the grid of square cells as a matrix and a live cell is indicated by placing the value 1 on the corresponding element in the matrix. Obviously, each cell in the grid has a neighbourhood consisting of the eight cells in every direction including the diagonals. We simulate the game of life over a limited time, on a grid of 10x10 with the help of the following model:

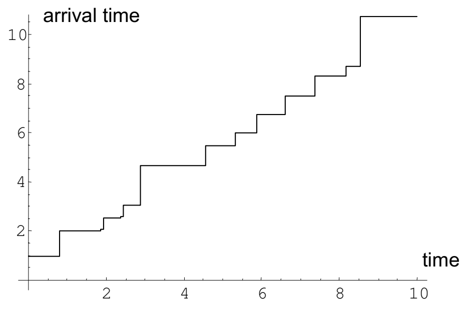


Fig. 7. Customers generated at random time moments

```
model GameOfLife
  parameter Integer n=10;
  parameter Integer initialAlive[:,2]=
                    {{2,2},{2,1},{1,2},{3,3}};
  discrete Integer lifeHistory[n,n](start=zeros(n,n));
equation
  initial equation
   lifeHistory=
      firstGeneration(pre(lifeHistory),initialAlive);
  when sample(0,0.1)
     lifeHistory =nextGeneration(pre(lifeHistory));
  end when;
end GameOfLife;
```

The simulation starts with an initial generation represented by the `initialAlive` parameter which updates the `lifeHistory` variable in the `initial` equation section. This is done by calling the function `firstGeneration`.

```
function firstGeneration
  input Integer M[:,:];
  input Integer I[:,:];
  output Integer G[size(M,1),size(M,1)]:=M;
algorithm
  for i in 1:size(I,1) loop
     G[I[i,1],I[i,2]]:=1;
  end for;
end firstGeneration;
```

At each event, generated by the sample statement, the `nextGeneration` function is called which applies the rules of the game to each cell of the grid. To apply one step of the rule we count the number of live neighbours for each cell. The value of the cell depends on the number of live neighbours according to the following rules:

– A dead cell with exactly three live neighbours becomes a live cell (birth).
– A live cell with two or three live neighbours stays alive (survival).
– In all other cases, a cell dies or remains dead (overcrowding or loneliness).

The rules of the life game are illustrated by the nextGeneration function:

```
function nextGeneration
  input Integer M[:,:];
  output Integer G[size(M,1),size(M,1)];
  protected
    Integer iW,iE,jN,jS,borderSum;
    parameter Integer n=size(M,1);
algorithm
  for i in 1:n loop
    for j in 1:n loop
      iW: = mod(i-2+n, n)+1; iE: = mod(i+n,n)+1;
      jN: = mod(j+n, n)+1;   jS: = mod(j-2+n,n)+1;
      borderSum := M[iW,j] + M[iE,j] + M[iW,jS] + M[i,jS] +
                   M[iE,jS] + M[iW,jN] + M[i,jN] + M[iE,jN];
      if borderSum ==3 then G[i,j]:=1;
      elseif borderSum==2 then G[i,j]:=M[i,j];
      else G[i,j]:=0;
      end if;
    end for;
  end for;
end nextGeneration;
```

4 Array Comprehensions in Modelica

Several powerful declarative language constructs are included in Modelica in order to strengthen the use of the language as mathematical specification language. The Modelica array comprehension language construct allows the selection of arbitrary dimension(s) for iteration where no dimension is treated preferentially and also allows simultaneous iterations over two or more arrays.

A general syntax for iterating over arrays is given below:

$$\texttt{expression for iterator}_1\texttt{, ...,iterator}_n$$

This expression will generate an array with `n` dimensions by evaluating the expression for each iteration variable value and collect the result into an array. For example, the expression {`i*i for i in 1:5`} evaluates to a vector of size 5: {`1,4,9,16,25`}. In a similar way the following special matrix with `1` at the diagonal and the rest of the elements equal to the sum of the row and column index:

$$\begin{pmatrix} 1 & 3 & 4 & 5 \\ 3 & 1 & 5 & 6 \\ 4 & 5 & 1 & 7 \\ 5 & 6 & 7 & 1 \end{pmatrix}$$

This array can be created using the following code.

```
{if i=j then 1 else i+j for i in 1:size(A,1),
                            j in 1:size(A,2) }
```

The array comprehension can be extended over the predefined Modelica functions: `max, min, sum` and `product`, which then act as reduction functions. Given the function $f_n(x) = \frac{n}{1+n^2}$ the sum $\sum_{i=1}^{N} f_n(x)$ can be expressed in array comprehension as: where `N` is a given natural number. Without array comprehensions the `sum` can be expressed by the following function:

```
function mySum
  input Integer N;
  output  Real result;
algorithm
  result:=0;
for i in 1:N loop
  result:=result+i/(i+i^2);
end for
```

5 Modelica Packages

Packages in Modelica may contain definitions of constants and classes including all kinds of restricted classes, functions, and subpackages. By the term subpackage we mean that the package is declared inside another package, no inheritance relationship is implied. Parameters and variables cannot be declared in a package. The definitions in a package should be related in some way, which is the main reason they are placed in a particular package. Packages are useful for number of reasons:

– Definitions that are related to some particular topic are typically grouped into a package. This makes those definitions easier to find and the code more understandable.
– Packages provide encapsulation and coarse-grained structuring that reduces the complexity of large systems. An important example is the use of packages for construction of (hierarchical) class libraries.
– Name conflicts between definitions in different packages are eliminated since the package name is implicitly prefixed to names of definitions declared in a package.

– Information hiding and encapsulation can be supported to some extent by declaring `protected` classes, types, and other definitions that are only available inside the package and therefore inaccessible to outside code.
– Modelica defines a method for locating a package by providing a standard mapping of package names to storage places, typically file or directory locations in a file system.
– Identification of packages. A package stored separately, e.g. on a file, can be (uniquely) identified.

As an example, consider the package `ComplexNumbers` which contains a data structure declaration, the record `Complex`, and associated operations such as `Add`, `Multiply`, `MakeComplex`, etc. The package is declared as encapsulated which is the recommended software engineering practice.

```
encapsulated package ComplexNumbers
  record Complex ...
  function Add ...
  function Multiply ...
end ComplexNumbers;
```

The example below presents a way how one can make use of the package `ComplexNumbers`, where both the type `Complex` and the operations `Multiply` and `Add` are referenced by prefixing them with the package name `ComplexNumbers`.

```
class ComplexUser
  ComplexNumbers.Complex  a(x=1.0, y=2.0);
  ComplexNumbers.Complex  b(x=1.0, y=2.0);
  ComplexNumbers.Complex  z,w;
equation
  z = ComplexNumbers.Multiply(a,b);
  w = ComplexNumbers.Add(a,b);
end ComplexUser;
```

Since classes can be placed in packages, and packages is a restricted form of class, Modelica allows packages to contain *subpackages*, i.e. packages declared within some other package. This implies that a package name can be composed of several simple names appended through dot notation, e.g. *"Root package"." Package level two"." Package level three"*, etc. Typical examples can be found in the Modelica standard library, where all level zero subpackages are placed within the root package called Modelica. This is an extensive library containing predefined subpackages from several application domains as well as subpackages containing definitions of common constants and mathematical functions. A few examples of names of subpackages in this library follow here:

```
Modelica.Mechanics.Rotational.Interfaces
Modelica.Electrical.Analog.Basic Modelica.Blocks.Interfaces
```

The equation-based foundation of the Modelica language enables simulation in an extremely broad range of scientific and engineering areas. For this reason an extensive Modelica base library is under continuous development and improvement being an intrinsic part of the Modelica effort (see www.modelica.org). Some of the model libraries include application areas such as mechanics, electronics, hydraulics , heat flow, etc.. These libraries are primarily intended to tailor Modelica toward a specific domain by giving modelers access to common model elements and terminology from that domain.

6 Acknowledgements

The authors would like to thank the other members of the Modelica Association for inspiring discussions and their contributions to the Modelica language design. Modelica 2.0 was released February 15, 2002. The Modelica Association was formed in Linkping Sweden, at Feb. 5, 2000 and is responsible for the design of the Modelica language (see www.modelica.org).

We also thank the entire MathModelica team from MathCore AB [12] and the Dymola team at Dynasim [10], as well as our colleagues at PELAB - the Programming Environment Laboratory [11], without whom this work would not have been possible.

References

[1] Abadi M. and L. Cardelli, *A Theory of Objects*, Springer Verlag, ISBN 0-387-94775-2, 1996.

[2] Elmqvist, H.; S. E. Mattsson and M. Otter. 1999. "Modelica - A Language for Physical System Modeling, Visualization and Interaction." In *Proceedings of the 1999 IEEE Symposium on Computer-Aided Control System Design* (Hawaii, Aug. 22-27).

[3] Fritzson, P.; P. Bunus "Modelica, a general Object-Oriented Language for Continuous and Discrete-Event System Modeling and Simulation." In *Proceedings of the 35th Annual Simulation Symposium* (San Diego, California, April 14-18, 2002)

[4] Fritzson, P. "*Introduction to Modelica*". First chapter of book draft. www.ida.liu.se/ pelab/modelica

[5] Fritzson P.; J. Gunnarsson; M. Jirstrand. "MathModelica - An Extensible Modeling and Simulation Environment with Integrated Graphics and Literate Programming." In *Proceedings of the 2nd International Modelica Conference* (18-19 March, Munich Germany, 2002)

[6] Fritzson P and V. Engelson. "Modelica, A Unified Object-Oriented Language for System Modeling and Simulation." In *Proceedings the 12th European Conference on Object-Oriented Programming*(ECOOP'98). (Brussels, Jul. 20-24,1998).

[7] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Tutorial and Design Rationale Version 1.4* (December 15, 2000). http://www.modelica.org

[8] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 1.4.* (December 15, 2000). http://www.modelica.org

[9] Tiller M. *Introduction to Physical Modeling with Modelica.* Kluwer Academic Publishers, 2001.

[10] http://www.dynasim.se - Dymola and Dynasim AB, Sweden.

[11] http://www.ida.liu.se/ pelab/Modelica - research site related to the Modelica language and the Modelica Open Source project.

[12] http://www.mathcore.se MathModelica and MathCore AB, Sweden.

The Return of Jensen's Device

Timothy A. Budd

Department of Computer Science
Oregon State University
Corvallis, Oregon, USA

Abstract. In the early 1990's my students and I developed Leda, a multiparadigm language based on the Pascal model. Leda allowed programmers to create abstractions in an object-oriented, functional, or logic programming style. More recently we have been interested in expanding this work, but this time using Java as the language basis. The objective to to add as few new operations as possible, and to make these operations seem as close to Java as possible, so that they seem to fit naturally into the language. To date we have implemented facilities for functions as first-class values, pass-by-name parameters, ML style pattern matching, and for relational (or logic) programming. In addition, we are currently evaluating a number of other extensions to the language.

1 Introduction

In the book *Multiparadigm Programming in Leda* [7], I described a multiparadigm programming language based on the Pascal model. My students and I purposely selected Pascal for our foundation language because we wanted a syntax that would be non-intimidating to most users, particularly beginning programmers. By adding only a few simple features (functions as values, nameless or lambda functions, relations) we were able to demonstrate how applications could be developed in a functional style, in a Prolog logic-programming style, or in an object-oriented style. Furthermore, a synergy quickly develops between the different paradigms, and a great deal of power derives from combining code written in two or more styles in a single program.

As Java has slowly become the dominant language in the academic world, we have become interested in the question of whether or not it would be possible to add similar features to Java. Many experiments in this area have already been tried. The best known multiparadigm version of Java is Pizza [21]. However, we felt that Pizza (and other similar languages) suffered from one fault that we had taken pains to avoid in the original design of Leda. This was that the additions did not remain true to what James Gosling calls the *feel* of the language [14]. With Pizza it is far too easy to tell when you leave Java and start programming in Pizza. We wanted something that was more subtle, so that the features seem to flow naturally into the existing Java syntax, without clear-cut boundaries. The reader will have to judge for themselves whether or not we have achieved our goal.

In this paper we will describe five major modifications we have introduced into J/mp, our extended Java language. These additions are:

– First class Functions. Functions as values, arguments, and results. Nameless functions created as expressions.
– Pass-by-name parameters. A technique for bidirectional information flow through parameters, and for delaying the evaluation of an argument until it is used (lazy evaluation).
– Operator overloading. Mostly just syntactic sugar used to reduce the size of a program and make it easier to read.
– Pattern Matching. We have extended the instanceof operator so that in addition to testing the dynamic class of an expression, it also breaks a value that was constructed using composition into its component parts.
– The relation as a data type, and Prolog-style relational programming. As we did with Leda, this feature is constructed by combining the above features with a library of operations that implement logic programming tasks such as backtracking and unification.

In the remainder of the paper we will describe each of these elements in turn.

1.1 Pass by Name

We began with the assumption that we would need to support functions as first class data types, similar to those found in Pizza. As with Brew [3], and unlike Pizza, the syntax we adopted is closer to the historical C/C++ model, and is to our eyes more in keeping with the feel of Java. Several examples will subsequently be presented.

The next observation was that we needed at least one more parameter passing mechanism, in addition to Java's approach of passing object references by value. In particular, for many of the applications we envisioned it was desirable to be able to pass information both into and out of a method or function by means of parameters, as well as a means to delay the evaluation of arguments. After considering many of the possibilities (for example, by-reference, copy in-copy out), we arrived at what might be a rather surprising conclusion. The parameter passing mechanism most suited to our purposes was an old and nowdays seldom used technique, pass-by-name [20].

In large part, this decision was influenced by the fact that pass-by-name fits quite easily into the object-oriented philosophy, and hence there is a simple technique for implementing the mechanism. We will describe this in Section 2.

The classic example used to illustrate the effect of pass-by-name parameters is *Jensen's device*, a procedure for taking a simple sum of a collection of values [11]. (Named after Jørn Jensen, who first explored its properties). Our first version of Jensen's device is shown in Figure 1, together with a class that illustrates how it can be invoked. The intent that the first and third arguments to the function Jensen are to be passed by name is indicated by the plus sign preceding the type. Each time the variable i is modified the corresponding actual parameter in the

```
public double Jensen(+int i, int max, +double x) {
    double sum = 0.0;
    for (i = 0; i < max; ++i)
        sum += x;
    return sum;
}

class Main {
    static double [ ] data = {1.5, 2.7, 3.2, 4.1, 5.2, 6.3};

    static public void main (String [ ] args) {
        int i;
        System.out.println("Sum " + Jensen(i, 6, data[i]));
        System.out.println("Sum of odds " + Jensen(i, 3, data[1+2*i]));
    }
}
```

Fig. 1. Jensens Device, Version 1

calling procedure will be changed. The fact that x is passed by name means that the evaluation of this parameter is delayed until the point where it is used, in the body of the loop. Furthermore, each time it is used, the actual argument is reevaluated.

The sample main program shows how this can be used to compute the sum of an array, or alternatively the sum of the odd-indexed elements in an array. By varying the parameters passed to Jensen's device a wide variety of behaviors can be produced [17]. For example, although seemingly generating only a one dimensional summation, a summation of a two dimensional array could be obtained as follows:

```
    Jensen(i, n, Jensen(j, m, a[i][j]));
```

Similarly, a dot product of two vectors could be computed as:

```
    Jensen(i, n, a[i]*b[i]);
```

1.2 Free Standing Functions

Figure 1 also illustrates the syntax used for free standing functions; that is, functions not associated with any class. As with Pizza, Brew, and similar systems such functions are ultimately first turned into an interface, and then into a class. Later in Section 2 we will show the transformation of this function.

1.3 Functions as Values

Many people dislike pass-by-name, although we view its bad reputation is being largely undeserved. Nevertheless, a number of the interesting things one can do with pass-by-name can also be accomplished using functions as first class values. (This is not surprizing. The implemenation of pass by name can in a certain sense be considered a special case of functions as arguments). Our second version of Jensen's device (Figure 2) illustrates this. Here there are two formal parameters, the first an integer and the second a function that takes an integer as argument and returns a double. The loop repeatedly invokes the function in order to yield different values.

```
public double Jensen(int max, double (int) f) {
   double sum = 0.0;
   for (int i = 0; i < max; i++)
      sum += f(i);
   return sum;
}

class Main {
   static double [ ] data = {1.5, 2.7, 3.2, 4.1, 5.2, 6.3};

   static public void main (String [ ] args ) {
      System.out.println("Sum of odds " +
         Jensen(3, double (int j) { return data[1+2*j];}));
   }
}
```

Fig. 2. Jensens Device, Version 2

The syntax used to define the type field of the function parameter largely mirrors the standard Java syntax for function prototypes, although the parameter names are omitted. The invocation (in the method main) illustrates the creation of a nameless function argument. Such arguments are often termed *lambda expressions*, a term adapted from the lambda calculus. Alternatively, a named function could have been passed as argument to Jensen, although not a named method (functions are first class values, methods are not).

A function that implements a curry illustrates the three common uses for function values; these are functions as arguments, functions as return types, and the creation of an anonymous function value:

```
int(int) curryLeft (final int left, final int(int, int) theFun) {
   return int(int right) { return theFun(left, right); };
}
```

Note that it has been necessary to declare the arguments as final in order for them to be captured in the context for the function. Also notice that the syntax for functions in J/mp requires no additional keywords or operators, and that, like Brew [3], our syntactic extensions are very similar to existing features in the Java language. The value curryLeft could itself be saved in a variable declared as follows:

```
int(int)(int, int(int, int)) c = curryLeft;
```

1.4 Infinite Length Lists

Functions as first class values have many interesting and unusual uses. A good example is the creation of lazy lists; lists that use lazy evaluation and hence do not calculate their elements until needed. Such lists can be used to represent collections that are infinite in length. A simple definition of a lazy list in which each element is computed as a transformation on the prior value can be written as follows:

```
class llist {
   private int hd; // current value
   private int(int) tf; // transformation function

   public llist (int h, int(int) t) { hd = h; tf = t; }

   public int head () { return hd; }
   public llist tail () { return llist(tf(hd), tf); }
}
```

Some would argue that this class is not only lazy but stupid, in that it reevaluates its elements each time a request is made. An alternative, more traditional, lazy list that delays evaluation until necessary, but thereafter remembers values that it has already computed, can be written but is not as concise. Using this class definition, the list containing all natural numbers can be generated as follows:

```
llist naturals = llist(0, int(int x) { return x+1; });
```

One feature to note is the lack of the new operator in this expression. If a class name is used in the fashion of a function, an implicit creation is assumed.

Slightly more complicated is the list of all prime numbers. A straightforward algorithm for computing these is the following:

```
llist primes = llist(2, int(int x) {boolean flag = true;
      while (flag) { flag = false; x++;
         for (int y = 2; (y * y <= x) && ! flag; y++)
```

```
        flag = (x%y==0);
    }
  return x; });
```

Many interesting effects can be achieved by manipulating infinite length lists, and developing functions that will filter or transform elements from such lists [7].

1.5 Pattern Matching

The language ML popularized a style of programming where compound data values were created by means of constructors, and then broken back into their constituent parts by means of pattern matching. The following ML definitions and function illustrate this behavior:

```
datatype Tree =
  Leaf of int
  | Node of int * Tree * Tree;

val t = Node(3, Node(4, Leaf(5),  Leaf(7)), Leaf(2));

fun   sum(Leaf(v)) = v
  | sum(Node(v, left, right)) = v + sum(left) + sum(right);
```

Constructors in Java serve much the same purpose as constructors in ML. This is particularly true given that we have made the new keyword optional, as previously described. A Java class definition similar to the ML code can be written as in Figure 3. Given this definition, we can construct a tree as follows:

```
Tree t = Node(3, Node(4, Leaf(5),  Leaf(7)), Leaf(2));
```

While constructors serve the same purpose in Java and ML, there did not seem to be a good substitute in Java for pattern matching. In pondering the possibilities, and remembering our goal of maintaining the spirit of Java and making minimal changes to the language, we determined that the closest operation to pattern matching in the existing language was the instanceof operator. By simply adding an optional argument list to this operator we could make it serve the dual purposes of type testing and deconstruction.

```
  t instanceof Node(value, left, right)
```

But how to provide semantics to the pattern matching operation? We initially sought to implement this operation outside of the class being tested. However, it is common for Java programmers to protect their data fields, as with the data fields value, left and right in Figure 3. Therefore a compound object represented as an instance of a class cannot easily be broken into its original parts outside

```
class Tree {
    protected int value;
}

class Leaf extends Tree {
    public Leaf (int v) { value = v; }
    public Leaf$ (+int v) { v = value; }
}

class Node extends Tree {
    private Tree left, right;
    public Node (int v, Tree l, Tree r) { value = v; left = l; right = r; }
    public Node$ (+int v, +Tree l, +Tree r) { v = value; l = left; r = right; }
}
```

Fig. 3. Class Definitions for a Binary Tree

of the class definition. For this reason we decided the author of the class must be directly involved in the pattern matching operation. A compound instanceof operator will internally be converted into a type test which, if successful, will invoke the deconstructor method. A deconstructor is written as the name of the class followed by a dollar sign, as shown in Figure 3. Typically the deconstructor uses by-name parameters to pass the values back to the operator invocation. (For an alternative approach to pattern matching in a Java extension, see [21].)

The following function computes the sum of the values in a binary tree, and illustrates the use of the pattern matching operations.

```
public int sum (Tree t) {
    int value;
    Tree left, right;
    if (t instanceof Node(value, left, right))
        return value + sum(left) + sum(right);
    else if (t instanceof Leaf(value))
        return value;
    return 0;
}
```

1.6 Operator Overloading

Each of the standard operators is assigned a textual name (plus for +, times for *, and so on). If the left argument to a standard operator is a class type, then a search will be performed in this class to see if a method has been defined using this textual name, and with a type signature that matches the right argument. If such a method is found then the operator is turned into a standard method invocation. We will see an example of this in the next section. This is the technique we used previously in Leda [7], as well as other languages, such as Python [4]. It is also in keeping with a proposed change to Java [13].

1.7 Logic Programming

The combination of by-name parameter passing and functions as values permits a simple implementation of logic programming. Students often have very limited exposure to this useful programming style; often viewing it as merely an obscure practice found only in marginal languages, such as Prolog [10]. Logic programming is sometimes termed *relational programming*, and a classic example is a database of family relations.

Logic programming in J/mp is provided using a supporting class named Relation. A relation can be thought of as a generalization of booleans, however the more accurate class definition is shown in Figure 4. The fundamental operation on a relation is apply, which takes as argument another relation, and returns a boolean which indicates that both the receiving relation and the argument relation can be satisfied.

Another fundamental operation with relations is the unification operator, named unify. Unify has two jobs. If both arguments are defined, it ensures that they are equal before testing the relation it is given. Otherwise, if the first argument is undefined, it is set to the second, and then the argument relation is tested. If the argument (the continuation) fails, the assignment is undone, and the unification fails. (Technically, this is not as general as the Prolog style unification, where either or both arguments can be undefined. However, it is sufficient for most purposes, and if desired the more general operation can be written using this simpler version as a basis).

An example to illustrate how these operations can be used is the following:

```
private static Relation eq(+String a, String b) { return Relation.unify(a, b); }

public static Relation progeny(+String f, +String m, +String c) {
   return (eq(f, "Albert") && eq(m, "Victoria") && eq(c, "George"))
   || (eq(f, "George") && eq(m, "Elizabeth") && eq(c, "Elizabeth"))
   || (eq(f, "Phillip") && eq(m, "Elizabeth") && eq(c, "Charles"))
   || (eq(f, "Charles") && eq(m, "Diana") && eq(c, "William"))
   || (eq(f, "Charles") && eq(m, "Diana") && eq(c, "Henry"));
}
```

Here the method eq simply provides a convenient shorthand for the invocation of the unification method. The method progeny is a typical logic programming database. It takes three by-reference parameters; representing a father, mother and child in the progeny relationship.

Queries in the logic programming style in J/mp are driven by either an if or a while statement. The conditional if seeks a single solution to a given query, while

```
class Relation {
    public boolean apply (Relation continuation) { return true; }

    public boolean asBoolean() { return apply(new Relation()); }

    public static Relation unify (+Object left, final Object right) {
        return new Relation () {
            public boolean apply (Relation continuation) {
                if (left == null) {
                    left = right;
                    if (continuation.apply(new Relation())) return true;
                    left = null;
                } else if (right != null) {
                    return left.equals(right) && continuation.apply(new Relation());
                }
                return false;
            }
        };
    }

    public Relation and (+Relation right) {
        final Relation me = this;
        return new Relation () {
            public boolean apply (final Relation continuation) {
                return me.apply(new Relation() {
                    public boolean apply(Relation r) {
                        return right.apply(continuation);
                    }
                });
            }
        };
    }

    public Relation or (+Relation right) {
        final Relation me = this;
        return new Relation () {
            public boolean apply (final Relation continuation) {
                return me.apply(continuation) || right.apply(continuation);
            }
        };
    }
}
```

Fig. 4. Definition of Relation

the looping while statement seeks all possible bindings. If the actual arguments are already defined they are used as a pattern, and if not defined they are set by the call on the relation. Assuming that a and b are variables of type String that are initially null, the name of a single child of Phillip, for example, could be determined as follows:

```
if (progeny("Phillip", a, b))
    System.out.println("child of Phillip " + b);
```

A characteristic of logic programming is that the same parameters sometimes serve as input and other times as output. For example, the father of Charles can be determined as follows:

```
if (progeny(c, d, "Charles"))
    System.out.println("father of Charles " + c);
```

A while statement can be used to cycle through all bindings of the argument values, as follows:

```
while (progeny("Charles", "Diana", e))
    System.out.println("children of Charles and Diana " + e);
```

New relations are easily defined using existing relations. The definitions for parentOf and grandParentOf, for example, are as follows:

```
public static Relation  parentOf (+String p, +String c) {
    String a; // either father or mother
    return progeny(p, a, c) || progeny(a, p, c);
}

public static Relation grandParentOf (+String g, +String c) {
    String a;
    return parentOf(g, a) && parentOf(a, c);
}
```

Using these one could discover, for example, the grandparent of William using a query such as the following:

```
if (grandParentOf(f, "William"))
    System.out.println("grand Parent of William " + f);
```

It is also easy to integrate conventional boolean expressions with relations, however the details are not shown here.

2 Implementation

Like Pizza and other similar systems [2, 3, 6, 12, 19, 21], J/mp is implemented as a source to source translation system. That is, J/mp programs are first processed into equivalent Java programs, which are then compiled using the standard Java system. In this section we will describe the most significant features of this transformation process.

As was done by the developers of the Pizza system, a function type is represented internally as an interface. A mangling algorithm is used to convert the function type signature into a unique name. (A technique that historically has been used in many different languages [15]). The following, for example, represents a function that takes an integer as argument, and returns a double as result:

```
interface double___int {
    public double call (int $0);
}
```

The simple signature encoding algorithm we use is adequate for primitive types, and for most class types, which are represented simply by the class name. It is legal for a Java program to contain two identically named classes, as long as they come from different packages. This would potentially cause problems with finding unique names, but such examples do not appear to arise in practice.

Similarly, one might believe there there could be a subtle interaction between inheritance and assignment of function values (the covarient and contravariant issue for method overriding but now appearing in a different guise). For example, assuming that class Child is a subclass of Parent, would it ever make sense to to assign a function with type signature void(Child) to a variable declared as maintaining void(Parent), or vice versa? Fortunately, it is easy to show that such an action, if allowed, could potentially result in non-detectable type errors. This being so, the fact that this behavior is ruled out by the simple expedient of type signature encoding is a benefit, and not a problem.

Using the conversion of function types to interfaces, the second version of Jensen's device (Figure 2) is translated internally into the following Java definition:

```
public  class Jensen  implements double___int_double___int  {
    public double call (int max, double___int f) {
        int i;
        double sum = 0.0;
        for (i  = 0; i <max ; i++)
            sum += f.call(i);
        return sum ;
    }
}
```

The use of the interface means that any function that takes an integer as argument and returns a double (that is, any class that implements the interface `double___int`) can be passed as argument to this function.

By-name parameters are implemented by means of an intermediary object that is responsible for accessing and setting a value. In the context of parameter passing such an object is traditionally known as a *Thunk* [16]. A portion of the definition of class Thunk is as follows:

```
class Thunk {
   public Object get() { return null; }
   public Object set(Object x) { return null; }

   public int getInt() { return 0; }
   public int setInt(int a) { return a; }
   public double getDouble() { return 0.0; }
   public double setDouble(double a) { return a; }
   ...//
}
```

(As an alternative we could have implemented different classes, one for each primitive data type, but instead elected to define just a single class for all values). Using this class, the implementation of our first definition of Jensens device (Figure 1) is as follows:

```
public  class Jensen  implements double___$int_int_$double  {
   public double call (Thunk i, int max, Thunk x) {
      sum  = 0.0 ;
      for (i.setInt(0); i.getInt()<max ; i.setInt(i.getInt()+1))
         sum +=x.getDouble();
      return sum ;
   }
}
```

The majority of work involved in passing parameters by-name is incurred on the calling side, which must create the Thunk:

```
class Main  {
   static double [ ] data  = {1.5 ,2.7 ,3.2 ,4.1 ,5.2 ,6.3 };

   static public void main (String  [ ] args ) {
      class Context$2 {
         int i;
      }
      final Context$2 context$3  = new Context$2() ;
      System .out .println ("Sum " + new Jensen  ().call (new Thunk() {
            public int getInt () { return context$3 .i ; }
            public int setInt (int ThunkVar$4) {
                return context$3.i  = ((int) (ThunkVar$4));
            } } ,6 ,new Thunk()  {
            public double getDouble () {
                return data [context$3.i];
            } }));
    }
}
```

An anonymous inner class is used to create the Thunk. Since anonymous classes in Java do not capture their complete surrounding environment [1] (that is, they are not true closures), it is necessary to create our own data type to maintain the context needed by the Thunk. For this purpose we define an inner class (here named Context$2) and replace all references to the local variable with references to the context. The anonymous Thunk class then redefines the appropriate get and set methods. If a non-assignable expression is passed as argument only the get method is defined, so that a set on the corresponding formal parameter will have no effect.

Nameless function values (lambda functions) are also implemented using the ability for Java to form anonymous inner classes. Like thunks, they must also capture their surrounding environment in a closure. The main function used in the second Jensen program, for example, is implemented as follows:

```
class Main  {
    static double  [ ] data  = {1.5 ,2.7 ,3.2 ,4.1 ,5.2 ,6.3 };

    static public void main (String [ ] args) {
        System .out .println ("Sum of odds " +new Jensen().call(3,
        new double__int () {
            public double call (int j) { return data[1 +2 *j]; }
        }));
    }
}
```

Pattern matching is implemented using the existing instanceof operator, type-casting and message passing. Deconstructors are converted into a method that returns boolean true, while a statement such as:

```
    t instanceof Node(value, left, right)
```

is converted into

```
(t instanceof Node) && ((Node) t).Node$(value, left, right)
```

The backtracking mechanism used by the logic programming system is provided entirely by the methods and and or, shown earlier in Figure 4. (A detailed explanation of how these less-than-obvious functions actually operate can be found in [7]). To convert a relation to a boolean in the context of an if statement, it is only necessary to append an invocation of asBoolean. In this fashion an if statement described earlier in the paper is translated into:

```
if (progeny(c, d, "Charles").asBoolean())
   System.out.println("father of Charles " + c);
```

(This is ignoring the creation of Thunks for the parameters c and d, which is a separate operation). The transformation of a while statement is more complex. The statement portion of the loop is transformed into a relation that always fails, which is then passed to the test portion. Since the relation fails, the backtracking system will automatically cycle through all possibilities. Ignoring the creation of Thunks, the while loop given in the earlier discussion of logic programming is implemented as follows:

```
progeny("Charles", "Diana", e).apply (new Relation() {
   public boolean apply (Relation w) {
      System.out.println("children of Charles and Diana" + e);
      return false;
      }});
```

An unfortunate side effect of this approach is that return statements cannot be used within such a loop, as the control would transfer out of the method inside the artificially generated nested class, and not out of the original surrounding function.

An alternative approach to implementing Prolog in Java is given by Engel [12].

3 Future Investigations

More than just a static language, we view J/mp as a framework for experimenting with language features. J/mp is implemented using the antlr compiler-compiler generation system[1]. J/mp files are first converted into the standard antlr tree-based representation; then transformations are performed at the intermediate representation level, before transforming the tree back into a conventional Java program. This allows for an extremely flexible and efficient system, making it easy to incorporate new changes.

A considerable percentage of the current effort involves improving the implementation. Two examples of this are final detection and closure elimination. Parameter values will be captured automatically in closures if they are declared

[1] See www.antlr.org.

as final, but programmers seldom bother with this modifier. If we can determine when a parameter can be so declared we can automatically insert the modifier. Similarly we need to refine our algorithm for determining when closures are necessary; the current algorithm is overzealous, creating them even when they are not needed.

In addition to finding novel uses for the programming mechanisms we have described in this paper, there are a large number of alternative programming features we are considering, to determine if they meet our requirements for utility and maintaining the feel of the language. These include the following:

– The addition of generics [6]. Generic classes will be included in the next release of Java (version 1.4). Our interest is in the effect of generics on free standing functions, and the use of bounded generics.
– Type inference. It is sometimes annoying to have to declare and type variables when they appear in expressions, particularly if they are assigned only once. So it would be useful if the system could in some cases infer the type of otherwise undeclared variables. One current modest proposal is to combine this feature with the final keyword, so that an assignment such as

  ```
  final x = expression
  ```

  would have a type determined from the expression part, and not need an explicit type declaration.
– Retroactive abstraction by on-the-fly generation of adapters (sometimes known as structural subtyping). Oftentimes it is necessary to combine two class hierarchies that have a similar interface, but do not explicitly share a common ancestor.

  ```
  class OpenLookObject {
      public void display () { ... }
      public int move (int x, int y) { ... }
  }

  class MotifObject {
      public void display () { ... }
      public int move (int x, int y) { ... }
  }
  ```

  This can be addressed by first creating an interface for the common operations:

  ```
  interface XWindowsObject {
      public void display ();
      public int move (int x, int y);
  }
  ```

We may not be allowed to change the original classes. However, when an instance of the original class is assigned to a variable declared as the interface, we can implicitly construct an adapter. That is, in place of:

```
OpenLookObject v = new OpenLookObject();
    ...
XWindowsObject xojb = v;
```

We can generate:

```
final OpenLookObject $save = v; // necessary to save context
XWindowsObject xobj = new XWindowsObject() {
    public void display () { $save.display(); }
    public int move (int x, int y) { return $save.move(x, y); }
    }
```

– Mixins. Conventional classes provide inheritance of behavior and polymorphic assignment (the assignment of a child value to a parent variable). Interfaces provide polymorphic assignment, but no inheritance of behavior. The third option is the inheritance of behavior, but no assumption of assignability. Such a feature is termed a *mixin* [5]. (A mixin is in this sense very similar to a private inheritance in C++ [8]). Our proposal is to add a third category of modifier to the class heading, incorporates. A class that incorporates from one or more classes will inherit behavior, but not be allowed any rights of substitution:

```
class A extends B incorporates C implements D {
    ... // inherits behavior from both B and C
}

B b = new A(); // allowed
D d = new A(); // allowed
C c = new A(); // not allowed
```

To implement this feature in current Java an instance of C would be created within each instance of A, and the invocation of methods inherited from C would be implemented in A using this object. Difficulties are dealing with constructors, and the possibility that A might want to override methods inherited from C. The latter can be handled using inner classes, although the details are somewhat complicated.

– Automatic boxing and unboxing. Primitive types (integer, real and so on) are not true objects in Java. Thus when objects are required (for example, in a data structure) such values must be wrapped in a special class, such as Integer. The language C# hides this requirement from the user, by automatically boxing primitive values as objects when necessary, and unboxing them back into primitives when no longer required [18]. We are exploring how difficult it would be to add this feature to Java.

– Multimethods. Although the receiver is used for dynamic dispatch in Java, it is the static type of argument values that determine the method binding in an overloaded method. If Child is a subclass of Parent, the following, perhaps surprisingly, will execute the parent method both times, instead of the child method in the second instance:

```
class Test {
    public void test (Parent p) { ... }
    public void test (Child c) { ... }

    public void main () {
        Parent a = new Parent();
        Parent b = new Child();
        test(a);
        test(b); // dynamic type has no effect
    }
}
```

Our proposal is to allow methods with compatible argument type signatures to be combined using the vertical bar (the or operator). Subsequent methods would omit any modifiers or return type, which are not allowed to change.

```
class Test {
    public void test (Parent p) { ... }
    | test (Child c) { ... }
    ...
}
```

The first method would be augmented with the addition of dynamic dispatching code, while the remaining methods are translated as normal. This allows subclasses to override the behavior of such methods, something that is not always possible in other schemes that have been proposed for handing multimethods [3, 9, 21].

– Open classes. Craig Chambers recently described a technique to help circumvent the problem that classes permit easy extension through the addition of new datatypes, while functions permit easy extension through the addition of new operations [9]. His technique allows the addition of new methods to existing classes, using a method heading syntax similar to the following:

```
    void Point.print() { ... }
    void ColoredPoint.print() { ... }
```

Dispatch on such functions is more complicated than either dispatch on regular methods or dispatch on functions, but appears to be manageable.

4 Conclusions

In this paper we have introduced J/mp, which is a multiparadigm extension to Java designed to support programming in the functional and logic programming styles, in addition to the object-oriented style of Java. As with our earlier language Leda [7], our goal in developing J/mp has been to support a natural syntax that requires minimal additions to the base language, and retains the important feel of the original language. Of course, "natural" and "feel" are subjective terms, and hence an evaluation of our success or failure will only come with experience in developing systems using our language extensions.

Currently, J/mp extends Java through the addition of the following facilities:

- Functions as first class values.
- By-name parameters.
- Operator overloading.
- Pattern matching.
- A library for relational (or logic) programming.

Current efforts involve work on the implementation, exploring novel uses for these language features, and exploring new language features that might expand the flexibility or expressiveness of the language, while maintaining the feel of the original Java.

Further information on J/mp, including source code and installation instructions, are available on the authors web site `http://www.cs.orst.edu/~budd/jmp`.

References

[1] Ken Arnold, James Gosling and David Holmes, *The Java Programming Language*, 3rd Edition, Addison-Wesley, Reading, MA, 2000.

[2] Jonathan Bachrach and Keith Payford, "The Java Syntactic Extender", Sigplan Notices, 36(11): 31-42, November 2001.

[3] Gerald Baumgartner, Martin Jansche, and Christopher D. Peisert, "Support for Functional Programming in Brew", *Proceedings of the 2001 workshop on Multiparadigm Programming with Object-Oriented Languages*, John von Neumann Institute for Computing, Jülich, Germany, 2001.

[4] David M. Beazley, *Python Essential Reference*, New Riders Publishing, Indianapolis, IN, 2000.

[5] Gilad Bracha and William Cook, "Mixin-Based Inheritance", *Proceedings of the 1986 OOPSLA—Conference on Object-Oriented Programming Systems, Languages and Applications*; Reprinted in *Sigplan Notices*, 25(10): 347–349, 1990.

[6] Gilad Bracha, Martin Ordersky, David Stoutamire, Philip Adler, "Making the Future Safe for the Past: Adding Genericity to the Java Programming Language," in *Proceedings of the 1998 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1998, reprinted as Sigplan Notices, 33(10):183-200, October 1998.

[7] Timothy A. Budd, *Multiparadigm Programming in Leda*, Addison-Wesley, Reading, MA, 1995.

[8] Timothy A. Budd, *An Introduction to Object-Oriented Programming*, 3rd Edition, Addison-Wesley, Reading, MA, 2002.

[9] Curtis Clifton, Gary Leavens, Craig Chambers, and Todd Millstein, "MultiJava: Modular Symmetric Multiple Dispatch and Open Classes for Java," in *Proceedings of the 2000 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2000.

[10] William F. Clocksin and Chistopher S. Mellish, *Programming in Prolog*, Springer-Verlag, Berlin, 1981.

[11] Edsger W. Dijkstra, "Defense of ALGOL 60," Communications of the ACM, 4(11):502-503, November 1961.

[12] Joshua Engel, *Programming for the Java Virtual Machine*, Addison-Wesley, Reading, MA, 1999.

[13] James Gosling, "The evolution of numerical computing in Java," Sun Microsystems Laboratories, `http://java.sun.com/people/jag/FP.html`.

[14] James Gosling, "The Feel of Java", talk at 1996 ACM Conference on Object-Oriented Programming Languages and Applications, 1996.

[15] Richard G. Hamlet, "High-level Binding with Low-Level Linkers," Communications of the ACM, 19:642-644, November 1976.

[16] Peter Zilahy Ingerman, "Thunks", Communications of the ACM, 4(1):55-58, 1961.

[17] Donald E. Knuth, "The Remaining Troublespots in Algol 60", Communications of the ACM, 10(10):611-617, 1967.

[18] Stanley B. Lippman, *C# Primer*, Addison-Wesley, Reading, MA, 2002.

[19] Sean McDirmid, Matthew Flatt, Wilson Hsieh, "Jiazzi: New-Age Components for Old-Fashioned Java," in *Proceedings of the 2001 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2001.

[20] Peter Naur et al. "Report on the Algorithmic Language Algol 60", Communications of the ACM, 6(1):1-17, 1963.

[21] Martin Ordersky and Philip Wadler, "Pizza into Java: Translating Theory into Practice", *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Paris, France, January 1997.

Towards Linguistic Symbiosis of an Object-Oriented and a Logic Programming Language

Johan Brichau[1], Kris Gybels[1], and Roel Wuyts[2]

[1] Programming Technology Lab,
Vakgroep Informatica,
Vrije Universiteit Brussel, Belgium
{johan.brichau, kris.gybels}@vub.ac.be
[2] Software Composition Group,
Institut für Informatik,
Universität Bern, Switzerland
wuyts@iam.unibe.ch

Abstract. Reflective systems have a causally connected (metalevel) representation of themselves. Most reflective systems use the same language to reason about their metalevel representation as the language that is used to reason about their domain. In *symbiotic reflection* a different language is used at the metalevel. The practical usability of this symbiotic reflection is enhanced if a *linguistic symbiosis* is accomplished that transparantly integrates both languages. Implementing such a linguistic symbiosis is relatively straightforward if the meta language and the base language share the same programming paradigm. The problem becomes far more complex when the paradigms differ. This paper describes the extension of the symbiotic reflective system SOUL with a linguistic symbiosis between a logic meta language and an object-oriented base language.

1 Introduction

Reflection is a technique that realizes flexible systems. This is because a reflective system can manipulate data that is a representation of its own computation (called causally-connected self-representation). As such, a reflective system can adapt its own computation. For example, truly reflective programming languages, such as Smalltalk or CLOS, allow to introduce new language features or change the internal workings of existing ones.

Typically, the causally-connected self-representation (CCSR) of a programming language is expressed in the same programming language. There also exist reflective languages in which the CCSR is expressed in a different programming language (e.g. Agora [5] and RbCl [2]). But this language still adheres to the same

[1] Johan Brichau and Kris Gybels are research assistants of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.)

programming paradigm as the base language (e.g. object-orientation) which allows for a relatively straightforward integration. This integration is called *linguistic symbiosis* [2], which means that program elements from one language can be used transparently in the other language, and vice-versa. It also means that one language is actually implemented in the other language, which enables mutual reflective capabilities.

In this paper, we describe the linguistic symbiosis between a logic and an object-oriented programming language. The linguistic symbiosis is particularly hard to achieve because these paradigms are fundamentally different. In general, object-oriented programs consist of objects, containing state, that communicate through messages. All control flow is explicitly programmed. Logic programs consist of rules that describe how a certain fact is true under some conditions. The control flow is implicit, i.e. the interpreter will prove that a certain fact is true by proving its conditions. Also, a message send is fundamentally different from a predicate call. A message always returns a single result and must always be provided with a fixed number of arguments while a predicate can return multiple results for multiple variables that were left unbound in the predicate call. Numerous object-oriented extensions to logic programming languages exist. Basically, these extensions enhance logic programming with modularization, inheritance and late binding but the overall paradigm remains logic programming. This is fundamentally different from extending an object-oriented language with a logic programming paradigm.

Wuyts previously presented a system in which symbiotic reflection was introduced between an object-oriented base language and a logic meta language [8]. In the resulting system (SOUL), the CCSR is expressed in a logic language and it has shown to be particularly useful [3, 4, 7, 6] because logic programming languages are better suited to express reasoning algorithms such as type inferencing, design pattern extraction, architectural conformance checking, etc. . . .

The current implementation of SOUL already allows multi-paradigm programming with the logic and object-oriented paradigms because values can be exchanged between programs written in the different paradigms. But SOUL does not introduce a linguistic symbiosis. We cannot invoke programs implemented in the object-oriented language in the same way as programs implemented in the logic language. To solve this problem, we introduce a complete linguistic symbiosis between a logic and an object-oriented programming language, and apply this to SOUL. This allows object-oriented programs to *transparently* use libraries of logic programs (designed for reflective programming) during method execution. The implementer of the library can then make use of the full power of the logic environment while the regular object-oriented programmer is not exposed to it.

This paper is organized as follows. In the next section, we describe the SOUL symbiotic reflective system and how SOUL and Smalltalk are cross-bound. Section 3 describes extensions to SOUL to accomplish a linguistic symbiosis between SOUL and Smalltalk. In section 4 we describe an example application and section 5 discusses the open issues.

2 SOUL

SOUL (Smalltalk Open Unification Language) is a logic programming language implemented in Smalltalk. But SOUL offers more than the ability to write Prolog-like logic programs: it supports the embedding of Smalltalk expressions in a logic program, which are executed as part of the inference process. These Smalltalk expressions can use logic variables and Smalltalk objects can be unified with logic variables. Furthermore, logic programs can be invoked from within a Smalltalk program by invoking a query. This section shows how this integration of Smalltalk and SOUL is accomplished. A more detailed discussion can be found in [7]. We first give a brief overview of how logic programs in SOUL differ from Prolog programs.

2.1 Differences with Prolog

The differences between Prolog and SOUL are mostly syntactic.

Variables in SOUL start with a question mark (e.g. `?var`).
Lists are enclosed in '`<`' and '`>`' (e.g. `<1,2,3,4>`).
Rules are written with the keyword '`if`' instead of a '`:-`' symbol.
Modules are used to encapsulate logic declarations (facts and rules). Each logic declaration belongs to a module and is only visible in the module where it is defined [1].
Querying other modules A rule in one module can invoke a query in another module using the '`.`' operator. For example, invoking a query '`myQuery(?x)`' in a module called '`Mymodule`', is written as follows: `if Mymodule.myQuery(?x)`

2.2 Cross-binding Smalltalk and SOUL

Smalltalk in SOUL Invocation of Smalltalk programs from within logic SOUL programs is accomplished through special constructs called *Smalltalk term* and *Smalltalk clause*. Furthermore, Smalltalk objects are treated as constants. Since everything in Smalltalk is an object, these special constructs are the only new kind of logic terms that are a direct result of the integration. We now describe each construct in more detail.

Smalltalk term This is a term that contains a Smalltalk expression enclosed in square brackets `[...]`. Each time the inference engine has to unify this term with another term, the expression is evaluated and the resulting Smalltalk object is used to complete the unification (such as binding it to a logic variable). As such, the real Smalltalk objects themselves can be bound to logic variables (how this is done, is explained later). Furthermore,

[1] Modules have no syntactic notation. How they are built is out of the scope of this paper.

the Smalltalk expression is allowed to contain logic variables. For example, the SOUL declaration '`object([ Object new])`' is a logic fact with a Smalltalk term as its single argument that contains an instance of `Object` (not just a notation for it).

Smalltalk clause This is a predication that is syntactically the same as a Smalltalk term, except that the embedded expression must evaluate to true or false. For example, the SOUL declaration '`smaller(?x,?y) if [?x < ?y]`' is a logic rule that uses a Smalltalk clause to compare the values of `?x` and `?y`.

Smalltalk object Objects can get bound to logic variables, as a result of the unification of a Smalltalk term with a logic variable. We extend the unification-scheme of SOUL to include Smalltalk objects such that they are treated as constants. This means that objects only unify with (free) variables and themselves.

generate/2 predicate The `generate/2` predicate is a predicate that decomposes a Smalltalk collection object into subsequent results of a variable (similar to what `member/2` does for logic lists). For example the query '`if generate(?x,[Smalltalk allClasses])`' returns many results for `?x`, i.e. all classes in the Smalltalk image. This is because the expression `Smalltalk allClasses` returns a collection object and the generate predicate subsequently binds `?x` to each element of this collection.

SOUL in Smalltalk Since SOUL is implemented in Smalltalk, Smalltalk programs can use the SOUL implementation to start logic queries. Without the transparent invocation mechanism presented later on, a SOUL query has to be invoked by sending a message to the class `SOULEvaluator` and iterate over the returned results after the evaluation. The following is part of a Smalltalk method containing the invocation of a query:

```
...
a := 1 + 2.
results := (SOULEvaluator eval:'if member(?x,<1,2,3,4>)'
                                    withArgs: #((x a)) ) allResults.
results succes ifTrue:[...] ifFalse:[...]
...
```

This example shows how a query `member(?x,<1,2,3,4>)` should be invoked where the value of the logic variable `?x` is the value of the Smalltalk variable `a`. When the query succeeds, the evaluator will return an instance of the `SoulResults` class indicating the success of the query and holding a collection of all successful variable bindings. All the values of these bindings get converted from logic terms to Smalltalk equivalents as follows so they can be easily manipulated by the Smalltalk programmer:

Smalltalk objects are trivially mapped onto themselves.
Logic constants are mapped onto a Smalltalk symbol.

Logic integers are mapped onto a Smalltalk integer.
Logic lists are mapped onto Smalltalk `OrderedCollections`.
Logic functor terms are mapped onto a special class `CompoundTerm`

When the query fails, the evaluator also returns an instance of the `SoulResults` class indicating that the query failed.

2.3 Symbiotic Reflection

The SOUL interpreter is a reflective system because it is implemented in Smalltalk and can thus use the Smalltalk meta-object protocol to investigate and adapt its own implementation. By allowing the use of Smalltalk objects in the SOUL language, the SOUL programmer also has access to this MOP and can reify every Smalltalk program and thus also SOUL itself. Hence, SOUL is in symbiotic reflection with Smalltalk. For example, the rules that reify Smalltalk classes to logic declarations are:

```
class(?x) if
  not(var(?x)),
  generate(?x,[Smalltalk allClasses]).

class(?x) if
  var(?x),
  [Smalltalk allClasses includes: ?x].
```

The first rule implements one possible usage of the `class/1` predicate where the variable `?x` is not bound to a value. Therefore, the rule will subsequently bind `?x` to a Smalltalk class. The second rule implements the case where `?x` is bound to a value and therefore, it will check if this value is a Smalltalk class.

Hence, we can invoke a query to gather all classes or invoke a query to check if a class `Symbol` exists:

```
if class(?x).
if class([Symbol]).
```

With the Smalltalk system reified in SOUL, the power offered by logic programming was used to express design patterns, programming conventions and software architectures, to name but a few [3, 4, 7, 6].

3 Towards Linguistic Symbiosis

A *linguistic symbiosis* [2] for Smalltalk and SOUL means that a Smalltalk program can transparently call a SOUL program as if it was a Smalltalk program and a SOUL program can transparently call a Smalltalk program as if it was a SOUL program. The result is that a meta programmer can use the full power of symbiotic reflection while a Smalltalk programmer can use the reflective facilities offered by SOUL without knowing that they are actually implemented in logic programs.

In the integration of Smalltalk and SOUL as described above, it is actually explicitly coded where a logic program and where a Smalltalk program is used:

SOUL to Smalltalk Calling a Smalltalk program from SOUL is made explicit through the use of a Smalltalk term or clause (using a Smalltalk term or clause).

Smalltalk to SOUL Calling a SOUL program from within Smalltalk is made explicit by sending a query message to the SOULEvaluator class.

To accomplish linguistic symbiosis between the logic and object-oriented languages, we have to map the main concepts of both paradigms on each other. Basically, we chose to map message sends in the object-oriented paradigm on queries in the logic paradigm. So, the invocation of a query should be the same as a message send and vice-versa. Therefore, we propose the following mapping:

1. Smalltalk classes <=> SOUL modules;
2. Smalltalk message sends <=> SOUL predicates;
3. Smalltalk collections <= SOUL query results.

In the following sections we explain each mapping in more detail.

3.1 Smalltalk classes and SOUL modules

The namespace of all Smalltalk classes and the SOUL namespace of all modules is combined into one namespace which is accessible from both the SOUL and Smalltalk environment. This results in a combined dictionary of (logic) modules and (object-oriented) classes. Smalltalk classes encapsulate methods, while SOUL modules encapsulate logic facts and rules. This difference becomes apparent when sending messages to or invoking queries on both Smalltalk classes and instances as well as SOUL modules.

3.2 Querying Smalltalk objects

In order to be able to transparently send messages from within a SOUL program, message sends should be expressed as queries. As a result, messages need a representation in the form of a predicate. Therefore, we define a straightforward mapping of messages to predicates. The predicate name is the message selector and the predicate arguments are the message arguments with an extra last argument for the return value of the message. For example, the SOUL query:

```
if Array.new:(10,?instance),
  ?instance.at:put:(1,2,?returnvalue)
```

corresponds with the message(s):

```
instance := Array new:10.
returnvalue := instance at: 1 put: 2.
```

In this example, an array of size 10 is created and stored in the variable `instance`. Afterwards, the integer 2 is stored at position 1 and the returning value of this message is stored in the variable `returnvalue`.

3.3 Sending messages to SOUL modules

Conversely, in order to be able to transparently invoke SOUL queries from within Smalltalk methods, a query should be expressed as a message send. As a result, SOUL predicates should have a Smalltalk message representation. Therefore, we define a mapping of logic predicates to Smalltalk messages, using Smalltalk's keyword messages.

Keyword messages consist of a selector where the arguments are interleaved with the keywords of the selector and keywords always end with a colon (e.g: `at: index put: anObject`). Because such keyword messages allow for a non-ambiguous mapping of logic queries to Smalltalk messages, we require that the names of all logic predicates use the signature of a keyword message. This means that the predicate name should consist of as many keywords as the predicate's number of arguments. Furthermore, this mapping also requires that the keywords are unique for each predicate that is defined in a module. The benefit of using keyword messages is explained later. Some example logic declarations that use this naming convention are:

```
add:with:to:(?x,?y,?result) if ...
method:inClass:(?method,?class) if ...
class:(?class) if ...
```

Of course, the problem is that a logic query can return multiple results for multiple variables, as opposed to a Smalltalk method, which always returns only one single result. Moreover, the same logic predicate can be used in multiple ways (i.e. with all arguments bound, all arguments unbound or only some of them bound). These problems are addressed in the following subsections.

Translating multiway predicates to messages To translate the multi-way property of logic predicates to Smalltalk, a logic module automatically understands a message for each way in which a logic predicate in this module can be used. As such, a single predicate in a logic module (possibly implemented by different logic facts and rules) corresponds to a set of Smalltalk messages that can be sent to the logic module. Because the name of a predicate uses a keyword message signature, the signatures of the corresponding Smalltalk messages can be easily derived from the name of the predicate.

The mapping of logic predicates to Smalltalk messages is most easily explained by considering how we would write the invocation of a particular predicate as a Smalltalk message. As explained above, we use a naming convention for predicates where the name consists of a keyword for each argument the predicate takes, much like for method selectors in Smalltalk. When we want to invoke a predicate by sending a message to a logic module, we simply concatenate the keywords to get the selector of that message, without intervening colons except when we want to bind a value to a logic variable. The keywords for the last arguments are omitted if their corresponding parameters are not bound. In case that all arguments are left unbound, only the first keyword is used as a message selector (without colon).

For example, the logic predicate `add:with:to:/3` in the `Arithmetic` module defines the addition relation and it can be invoked in multiple ways. Therefore, the `Arithmetic` module understands the messages shown in table 1, where their corresponding query is also shown.

Message	Query
`add: 1 with: 2 to: 3`	`if add:with:to:(1,2,3)`
`add: 1 with: 2`	`if add:with:to:(1,2,?res)`
`add: 1`	`if add:with:to:(1,?y,?res)`
`add`	`if add:with:to:(?x,?y,?res)`
`addwith: 2`	`if add:with:to:(?x,2,?res)`
`addwithto: 3`	`if add:with:to:(?x,?y,3)`
`addwith: 2 to: 3`	`if add:with:to:(?x,2,3)`
`add: 1 withto: 3`	`if add:with:to(1,?y,3)`

Table 1. Mapping a multi-way predicate to messages

We further illustrate this with two example queries and their corresponding Smalltalk messages:

Calculate the addition of two numbers:

```
if Arithmetic.add:with:to:(1,2,?result)
     ?result -> 3

result := Arithmetic add: 1 with: 2.
```

Calculate the first argument of the addition, given the second argument and the result:

```
if add:with:to(?x,2,3)
     ?x -> 1

x := Arithmetic addwith:2 to:3.
```

A consequence of this approach is that for each message, we can document how many variables it returns (i.e. all arguments that were left unbound in the corresponding query of that particular message). The examples above use messages that return a single result for a single variable, but in table 1 many possible messages need to return more than one results for more than one variable.

Returning multiple variables When a message send leads to the invocation of a query that returns the binding of more than one variable, the results are returned in a Smalltalk `OrderedCollection` instance. This is not a break in our linguistic symbiosis, as we can document what the return result of that particular message is. For example the logic query to calculate the first two arguments of the addition that results in 3:

```
if add:with:to(?x,?y,3)
      ?x -> 1
      ?y -> 2
```

corresponds with the Smalltalk program:

```
| xyCollection |
xyCollection := Arithmetic addwithto:3.
```

In this example, two variables need to be returned, which means that a Smalltalk collection is returned containing the required `x` and `y` value. But we only showed one possible pair of results for `?x` and `?y`, while there are many more possible results (such as `?x -> 2 ,?y -> 1`).

Returning multiple results for each variable The returning of multiple results for each variable is a more complex part of the linguistic symbiosis. When a logic query also returns more than one result for each variable it returns, we can decide to either hide this from the Smalltalk programmer or explicitly return them in a collection. When returning all subsequent results as an explicit collection, the semantics are clear, but the disadvantage is that the Smalltalk programmer most probably experiences that he is actually invoking a SOUL query here.

For example, consider the following Smalltalk program that invokes the `add:with:to:/3` logic predicate and prints the results for `x` on the Transcript.

```
xyColl := Arithmetic  addwithto: 3
xColl := xyColl first.
xColl do:[ :x | Transcript show: x]
```

Because the subsequent results for `x` are returned in an explicit collection, the Smalltalk program has to explicitly enumerate over the results. Another solution we have experimented with, is to hide the collection from the Smalltalk programmer and represent it as a single result. The return value appears as a single result to the programmer, but it actually represents all subsequent results. Internally, this means that we do use a kind of collection but when a message is sent to this 'implicit collection', the message is automatically dispatched to all values of the collection. For example, using the 'implicit collection' solution, we can write the above program as follows:

```
xyColl := Arithmetic addwithto: 3
x := xyColl first.
Transcript show: x
```

Mind that if the program above would be used with the 'explicit collection' solution, the collection `#(0 1 2 3)` will be printed to the Transcript. But, while the 'hidden collection' provides us with the desired result in many cases, it also leads to confusing and erroneous behaviour of Smalltalk programs since messages with side effects are executed multiple times. An improvement for this solution is to be researched. For now, we continue using the first solution with explicit collections.

4 Practical Use

In this section we present an example of a system that is implemented using both logic and OO programming. The example is that of an e-commerce system in which prices of products are adapted to take into account reductions granted to a specific user of the system. Which reductions are applicable typically depends on a number of things, such as the customer's history with the company, the use of e-coupons etc. Logic programming lends itself well to expressing the rules governing the applicability of reductions. In her master's thesis, Goderis describes an architecture for e-commerce systems that allows knowledge such as reduction rules to be described using logic programming while the rest of the system is implemented in OOP [1]. In this work she identified a need to be able to easily interchange information and control between SOUL and Smalltalk programs. The linguistic symbiosis mechanism we presented in this paper should fulfill this need.

For this example we will consider a simple e-commerce system with the two important classes `Product` and `Customer`. The message `price` can be used on an instance of `Product` to retrieve its price.

What we want to do now is apply changes to the return value of the `price` method to reflect price reductions granted to the current user of the system. Such an adaptation is typically done in Smalltalk using object wrappers. We have similarly defined a *logic module wrapper*, which is simply a special kind of logic module that can be used as a wrapper around an object. Any message sent to the wrapper is forwarded to the wrapped object unless the message is understood by the module in the sense that it defines a predicate that maps to the message.

The wrapper for `Product` instances we need here defines the predicate `price/1` as follows:

```
price(?reducedPrice) if
  ?wrapped.price(?basePrice),
  findall(?reduction, reduction(?reduction), ?reductions),
  reduced(?basePrice, ?reductions, ?reducedPrice)
```

The variable `?wrapped` is defined by the wrapper module as referring to the wrapped object. Here it is used to get the answer to the price message from the wrapped `Product` instance.

Several definitions for the reduction predicate can be given, we give a simple example here:

```
reduction(10) if
  customer(?customer),
  ?customer.age(?age),
  greaterThan(?age, 65)
```

The above rule expresses that a special "senior's reduction" of 10% is applicable to customers aged 65 or older.

Furthermore, the wrapping of instances should occur at run-time. The Smalltalk reflective facilities offer this functionality. The following Smalltalk method runs periodically and wraps products when they are eligible for reductions:

```
activateReductions
     allProducts do: [:product |
            (knowledgeBase eligibleForReduction: product)
              ifTrue:[product wrap: ReductionWrapper new] ]
```

Using the symbiotic reflection and our linguistic symbiosis, it is now easy to describe when a product is actually eligible for reduction. The following logic rules do exactly this and are called by the Smalltalk program above.

```
eligibleForReduction:(?product,?result) if
     ?product.kind(toys),
     Date.today(?date),
     ?date.month(November).
```

The rule above describes that toys feature a price reduction in November.

5 Discussion and Future Work

In this paper, we describe ongoing work about the linguistic symbiosis of SOUL and Smalltalk. Because SOUL is a logic meta language over an object-oriented base language, this symbiosis requires a mapping of modules to classes and messages to queries. The advantage of this symbiosis is that the base programmer can use the meta programs as if they were implemented in the base language. Furthermore, it also provides us with the opportunity to optimize parts of the logic meta programs by transparently replacing them with Smalltalk meta programs. Besides enhancing the practical use of symbiotic reflection [8], the linguistic symbiosis can also benefit non-reflective programming. Programs that do no reflective programming can also make use of the power of the logic programming language. As such, through our linguistic symbiosis, we have also introduced multi-paradigm programming in Smalltalk with the object-oriented and logic paradigms. We have already tackled many issues in this linguistic symbiosis, but others still remain open. We now summarize the most important results and elaborate on the open issues.

5.1 Results

Trivially, to obtain a linguistic symbiosis, a shared namespace for both languages should exist to allow access to the global entities of both languages. In our case, this namespace contains references to all classes and logic modules. Because these entities should look and feel the same in both languages, we need to define a mapping between them. In our particular case this means that we need to map predicates onto messages. This mapping has a syntactic aspect and a semantic aspect.

The syntactic mapping defines the common look, while the semantic mapping is concerned with the common feel. The syntactic mapping is rather language-specific, while the semantic mapping is paradigm-specific. Indeed, when using another object-oriented language (e.g. Java), the semantic mapping of one multi-way logic predicate to a set of methods will remain while a totally different syntactic mapping will be needed. Furthermore, we return multiple variables from a logic predicate in a container. The returning of multiple results for each of those variables can also correspond to returning them in an explicit container. Another solution we proposed is to hide this, but the implication of such a mapping is still to be researched.

5.2 Open issues

Multi-way methods? The linguistic symbiosis, as defined above, treats methods as 'uni-way' logic predicates. An issue that remains to be defined is how a method can correspond to a multi-way logic predicate. Clearly, this is not a trivial issue and it will most likely involve the implementation of a method for each way in which the predicate can be called. We can envision a system in which a group of methods that is implemented according to a pattern gives rise to a single predicate, much in the same way as a logic predicate is mapped onto multiple methods. On the other hand, there also exist a reasonable amount of logic predicates that are not multi-way. So maybe this trade-off would be acceptable.

Multiple Results? As we already mentioned, a logic query can return multiple results for multiple variables. These multiple results can be hidden from the object-oriented programmer but this can lead to strange behaviour of object-oriented programs because messages will get executed several times.

Backtracking of side-effects? Even more complex issues arise when backtracking occurs in the logic program and methods that perform side effects have been executed. Of course, this is also true in pure logic programs that use logic predicates that execute side-effects.

Cross-language Inheritance? An issue that we did not mention at all, is how the inheritance relation could be implemented between logic modules and object-oriented classes. We consider this topic as future work.

5.3 Paradigm leaks

The above discussed problems of 'multiple results' and 'backtracking of side-effects' can be collectively described as 'paradigm leaks'. The symbiosis mechanism creates a leak where rules of one paradigm end up in the programming language based on the other paradigm. Dealing with and backtracking over multiple results in Smalltalk is not something considered to be part of the object-oriented paradigm, while side-effects are normally to be avoided in logic programming. Whether or not this leak is an undesirable effect in some cases remains to be investigated.

References

[1] S. Goderis. Personalization in object-oriented systems. Master's thesis, Vrije Universiteit Brussel, 2001.

[2] Y. Ichisugi, S. Matsuoka, and A. Yonezawa. Rbcl: A reflective object-oriented concurrent language without a run-time kernel. In *Proceedings of International Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-Level Architecture*, pages 24–35, 1992.

[3] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. In *Proceedings of the 13th SEKE Conference*, pages 236–243. Knowledge Systems Institute, 2001.

[4] T. Mens and T. Tourwe. A declarative evolution framework for object-oriented design patterns. In *Proceedings of Int. Conf. on Software Maintenance*. IEEE Computer Society Press, 2001.

[5] W. D. Meuter. The story of the simplest mop in the world - or - the scheme of object-orientation. In *Prototype-based Programming*, pages 24–35. Springer-Verlag, 1998.

[6] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS-USA '98*, 1998.

[7] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation.* PhD thesis, Vrije Universiteit Brussel, 2001.

[8] R. Wuyts and S. Ducasse. Symbiotic reflection between an object - oriented and a logic programming language. In ECOOP 2001 International workshop on Multi-Paradigm Programming with Object - Oriented Languages, 2001.

Replacing Refinement or Refining Replacement?
A Unifying Framework for Object-Oriented Methods

Sibylle Schupp

Dept. of Computer Science
Rensselaer Polytechnic Institute (RPI)
Troy, NY
schupp@cs.rpi.edu

Abstract. In object-oriented programming, one distinguishes between *replacement semantics*, where a child method overrides the corresponding parent method, and *refinement semantics*, where a parent method is specialized by the child one. Although from the standpoint of software development it would be advantageous to have both semantics available, none of the current object-oriented languages gives users a choice. In this paper we therefore investigate how a program can be designed that simulates replacement and refinement semantics in an ordinary user program, i.e., without language support. We provide a unifying view of refinement and replacement, in which replacement can be considered as a special case of refinement, characterize in an abstract, language-independent way the metaprogram that allows changing the predefined method semantics with one a user can control and present a prototype implementation, a framework in C++.

1 Introduction

One of the often quoted advantages of object-oriented programming is the combination of code reuse and extensibility. If users of a large object-oriented system need to augment the system, they need not start from scratch but extend the inheritance hierarchy incrementally. What portion of code can be reused, however, depends. While at object level every child incorporates its parent(s) and in that sense reuses its data members, the semantics of methods differs between the so-dubbed American school and the Scandinavian school of object-oriented programming: the former implements *replacement semantics*, the latter *refinement semantics* [2]. In languages of the Scandinavian school, e.g., Simula or Beta, it is therefore possible to inherit and successively refine method bodies. In contrast, in Java, C++, and other languages of the American school, designers of a derived class have to accept a method as it is implemented by the parent class, or else they have to entirely rewrite it. On the other hand, replacement semantics adds the freedom to entirely override a parent method, for example with a more efficient implementation, while in refinement semantics each generation only keeps adding code. In short, either semantics has its advantages over the other one, and either its disadvantages. The working programmer, therefore, would be served best having both semantics available. However, no language we are aware of supports replacement and refinement semantics at the same time.

If not directly supported by a language, could refinement semantics, then, be simulated by a program on top of it? Likewise, could replacement semantics be made available in a language that has no support for it built in? In other words, can a user program be designed that can take control over method dispatching and implement its own semantics? Organizing an issue that typically is part of the language implementation, such program certainly has to be a metaprogram that understands how to utilize and manipulate the existing compiler so that the compiler performs the desired steps automatically. Yet, how realistic is it to expect an object-oriented language to provide the means for a user program to cancel out predefined behavior?

In this paper we present a metaprogram that allows overriding the predefined method semantics with one a user can control. Given a method and an inheritance hierarchy, a user can decide at invocation time whether this method replaces or refines the corresponding ones in the base classes. Moreover, standard features of object-oriented languages suffice to realize this metaprogram, with one exception, though: the availability of a lazy binding mechanism seems to be necessary. Our current implementation is in C++, its core a parameterized, compile-time recursive method crawler based on the techniques of template metaprogramming [4]. Interestingly, template metaprogramming, which originally was introduced to emulate different programming styles in C++, thus can be used to emulate object-oriented programming in itself.

This paper makes three contributions. First, we provide a unifying view of refinement and replacement by defining refinement as a binary relation of methods in which replacement can be considered as just a special case of refinement. This definition suggests the design of a metaprogram that allows refinement to be specialized to replacement. Second, we characterize the metaprogram in an abstract, language-independent way so that it can be decided for an arbitrary object-oriented language whether or not it allows for an implementation of such metaprogram. Third, we discuss our prototype implementation, which is organized as a framework consisting of three parts: the metaprogram that handles method dispatching; the framework class that factors out tasks common to all user-defined classes and encapsulates them in a separate base class that the framework or library designers provide; and a certain programming discipline a user has to exercise when designing new classes and the main program.

We begin the presentation with a formalization of the notion of refinement and replacement in Sect. 2. For readers less familiar with the "Scandinavian semantics" Sect. 3 summarizes some applications of *action specialisation*, as it is called there, and gives an idea of the "look-and-feel" of the Beta language, the language that implements refinement idiomatically. Sect. 4 then identifies in an abstract sense what the requirements are for a metaprogram that can override method semantics. In the second part of the paper we show how these abstract requirements map to C++ features (Sect. 5), then turn to our C++ implementation and explain the framework in some detail. We first discuss the user interface and demonstrate the framework with an example program (Sect. 6), then present the framework class and the metaprogram (Sect. 7); the source code of the framework, including the metaprogram, is small enough to be completely listed in the appendix. An evaluation of the C++ metaprogram and the approach of user-controlled method semantics concludes the paper.

2 Definitions

Let $\mathcal{L}$ be a language that supports classes and inheritance.

Simplifying assumptions. Since type checking issues are irrelevant in our context we bypass all questions of coercion, contra- and covariance and refer to a method by its method identifier only. We furthermore refer to the direct base class of a class as a *parent* class and assume that each child has a unique parent to which a method f can be passed on if f is not defined in the child class itself. For a language with multiple inheritance the notion of the unique parent refers to the parent that is the legal receiver of the method. Without loss of generality, we assume that this parent is uniquely defined and, if not, that the language translator deals with the name clash implied. Finally, we identify a method body with the sequence of statements or expressions it comprises and omit its declarative sections.

Notation.

1. Let A, B be two classes. If A is a base class of B we write
$$B \prec A.$$
Obviously, $\prec$ is a transitive relation. Its reflexive closure is denoted by $\preccurlyeq$ and includes the class B itself.
2. We use subscript notation to indicate the class of the receiver, e.g., f_A means that a message f is sent to an instance of class A. Note that a class does not have to define a method to be a legal receiver of it. If it defines the method, we use $\equiv$ to bind the method identifier to its body:
$$f_A \equiv \langle a_1, \ldots, a_n \rangle,$$
where $a_1, \ldots, a_n$ are statements and expressions in $\mathcal{L}$, possibly encapsulated in other member functions of A.
3. Let e_1, e_2 be two statements or expressions in $\mathcal{L}$. If e_1 is executed before e_2, we write
$$e_1 < e_2.$$
The relation $<$ is transitive and irreflexive.

Definition 1. *Denote by $\mathcal{C}$ and $\mathcal{F}$ two sets of class and function identifiers, respectively. Let A, D be two classes in $\mathcal{C}$ with $D \preccurlyeq A$ and let f be a method defined in D and in each of its base classes:*

$$f_I \equiv \langle pre_I, post_I \rangle \quad \forall\, D \preccurlyeq I \preccurlyeq A,$$

where pre_I and $post_I$ are expressions or statements, possibly encapsulated in other members of I; we call pre_I and $post_I$ the prefix *and* postfix *of f_I. Let $(f, C, D) \subseteq \mathcal{F} \times \mathcal{C} \times \mathcal{C}$ be a relation such that*

1. $D \preccurlyeq C \preccurlyeq A$
2. $pre_I < post_I \quad \forall\, D \preccurlyeq I \preccurlyeq C$
3. *It holds*

$$pre_J < pre_D \wedge post_D < post_J \quad \forall\, J : D \prec J \preccurlyeq C,$$

Then (f, C, D) is called a refinement relation *and we say f_D* refines *f_C.*

Definition 2. *Let f, A, C, D be as in Def. 1. In the special case of $C = D$ we call the relation (f, D, D) a* replacement relation *and we say f_D* replaces *$f_I\ \forall\, D < I \preccurlyeq A$.*

With Def. 2, replacement and refinement are theoretically unified so that there is no need to distinguish between both. Unless said otherwise, the notion of refinement in the following always includes replacement as a special case.

3 Action Specialization in the Beta Language

As already discussed in the introduction, there are equally good reasons for the “Scandinavian” and the “American” semantics. However, since in the commercial world languages with replacement semantics predominate, it might be useful to recall some of the motivations for the use of refinement in the traditional sense. In this section we discuss some applications in which refinement semantics is advantageous and furthermore give an idea of the “look-and-feel” of the Beta language, the language that implements refinement in a paradigmatic way. In languages with a special semantics for constructor functions, some of the typical applications for refinement can be encapsulated in constructor functions.

3.1 Applications

The benefits of refinement semantics include a reduced amount of code a user has to provide as well as increased safety and stability of a program. The following list illustrates four typical kinds of applications.

– Resource management often requires the nesting of functions that “come in pairs.” Allocation, for example, requires the subsequent deallocation; and getting a file handle requires returning this handle. Often, it is the user who is responsible for the proper release of a resource—and often users forget to do so. Refinement semantics provided, a library or framework writer can take the burden of proper resource management away from a user by entirely factoring out this task and encapsulating it in a method of a separate class.

– The standard textbook case study of a booking system often is modeled as an abstract class that provides the interfaces to database operations and their partial implementation, but defers the full implementation to the specific (flight, theater, etc.) booking system. Here, refinement semantics allows defining the more specialized method incrementally, as the sequence of statements that extends the more general implementation in the base class, without the need to repeat the code defined there.
– Similar examples provide hierarchically structured documentation languages like XML where every non-empty element starts with a *start-tag* and ends with an *end-tag* and "the logical and physical structures must nest properly, as described in 4.3.2 Well-Formed Parsed Entities" [14]. If, for example, a company wants to give all its publications a uniform appearance while still leaving room for departments to have their own style, then refinement can help combining uniformity and extensibility: the root class can provide the first level in the hierarchy of the XML document, including the document type description, DTD, that renders an XML document as valid. Subsequent classes then inherit the overall document structure but can design the subsections still on their own.
– Finally, the original motivation for Simula to introduce refinement comes from the transfer of control from f_B to f_A and the, with that respect, symmetric relation between f_A and f_B. Since both f_B "calls" f_A and f_A "calls" f_B, either one suspends its execution in favor of the other one and resumes its execution at exactly the point of suspension. The two methods therefore are similar to coroutines. As such can they be applied to various simulation tasks, taking turns in playing the active and passive phase of the simulation.

3.2 Constructor semantics

In many languages with replacement semantics, constructors, and, for that matter, destructors, are treated specially and in fact are subject to a semantics similar to refinement semantics. Some of the examples that illustrate the benefits of refinement semantics can therefore be modeled by an appropriate constructor design. For example, the resource-acquisition-is-initialization idiom [13] suggests encapsulating the acquisition of a resource in the base class constructor, its release in the corresponding destructor. This idiom thus is based on the mechanism that the call to a child's constructor internally includes a call to its base class constructor, and that a call to its destructor recursively calls the destructors of the base classes.

3.3 The Beta Language

The first language supporting refinement, although in restricted form, was Simula [12, 9]. In Simula, there exist *prefix classes*, that are classes that allow for concatenation at different levels. Since classes can contain statement lists, subclassing a prefixed class means concatenating these statement lists and to invoke the statements of the prefix class before invoking the ones from its subclass. The keyword `inner` was introduced to allow the user to override the default order and to specify the invocation point of the statements of the subclass.

The Beta language extends Simula's idea of specialization. In Beta, everything is centered around the notion of a pattern, which syntactically unifies classes and functions [8, 6]. A pattern Q is defined as a pair of name and descriptor

```
Q: (# E #);
```

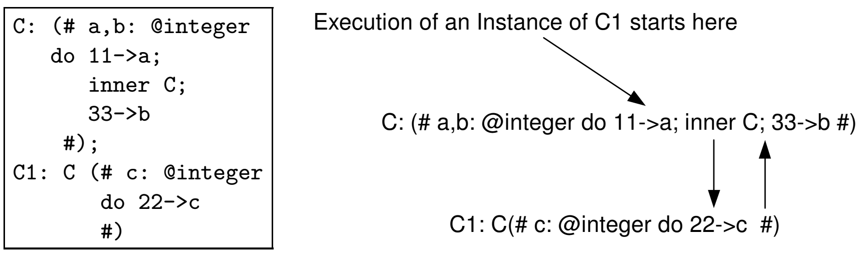
where `(#` and `#)` mark the beginning and end of the descriptor. Syntactically, there is no difference between a class pattern and a procedure pattern; of course, the descriptor of a class pattern can include procedure patterns, similar to class methods elsewhere. (Class and procedure) patterns contain an *action part*, which is further divided in three parts: *enter-part, do-part, exit-part.* The enter- and exit-part describe the input and output parameters, respectively, while the do-part consists of the *imperatives*, e.g., assignments, expressions, or control structures, that perform the action. A class pattern P can be subpattern of Q

```
P: Q (# E #);
```

and, likewise, can subpatterns be used for procedure patterns. In particular is it possible for a subpattern to specify additional imperatives. As in the case of Simula, the keyword `inner` specifies how the action part of the superpattern is combined with the one of the subpattern; if a descriptor has no subpattern, `inner` has no effect.

```
Q: (# E do E inner E #);
P: Q(# E do E #);
```

By default, `inner` refers to the immediate enclosing object descriptor; any other object descriptor can be given explicitly.



```
C: (# a,b: @integer
   do 11->a;
      inner C;
      33->b
    #);
C1: C (# c: @integer
       do 22->c
       #)
```

Execution of an Instance of C1 starts here
C: (# a,b: @integer do 11->a; inner C; 33->b #)
C1: C(# c: @integer do 22->c #)

Fig. 1. Example and illustration from the Beta book [8]: "The execution of a `C1` object starts by execution of the imperatives described in `C` (. . .) [and] implies execution of `11->a`, `22->c` and `33->b`." (Fig.6.5, Sect. 9.4).

4 Abstract Requirements

Assuming an (arbitrary) object-oriented language, how does a metaprogram in this language look that controls method semantics from the user level? In this section we abstractly identify the subtasks of such metaprogram and relate them to language features. As it turns out, all but one are standard features of object-oriented languages.

The absolute prerequisite for such metaprogram to work, informally speaking, is the availability of some kind of lazy binding mechanism. Given Def. 1, lazy binding is necessary since the execution of a method in a base class is interleaved with the one in the child class, which, however, does not yet exist when the base class is defined. Were every method at the base class level completely defined at class design time there were no way to later "insert" the method of a child class and to establish the execution order required. Lazy binding, however, allows (partially) deferring the definition of a method, thus can serve as a hook for the metaprogram to extend a method in the desired way.

Deferring means transferring control, to the metaprogram that organizes the method lookup. Transferring requires on the other hand the control to get back from the metaprogram to the refining class methods, to call the *prefix* defined in the next-lower level of the inheritance hierarchy and, ultimately, to return to the *postfix* defined at the current level. A callback mechanism between the metaprogram and the class hierarchy is necessary, therefore. The callback requires at the same time a notion of an order or sequential execution that guarantees, e.g., that one prefix has terminated before the subsequent one starts. This callback also has to be parameterized so that at every level of the inheritance tree the parent class can initiate the respective prefix call on the child. Given such parameterization, however, the special case of replacement can simply be realized by binding parent and child class to the same type.

Third, the interplay of suspending and resuming methods in the child and the parent class has to be implemented with object-oriented discipline. With no particular compiler assistance available, invoking the prefix of a base class method from within a child method requires getting from a child object to its parent, as this object is the proper one its method can be called on. Similarly, the suspension of the parent method and the resumption of the child method requires getting from the parent to the child object, as this time the child is the proper receiver. To be able to switch between child and parent object, the metaprogram needs to "traverse" the objects in the inheritance hierarchy, both upwards from child to parent and downwards, from parent to child.

Finally, the whole metaprogram must guarantee that no new instances are created of the object the methods of which are subject to the controlled semantics, in particular not of the object the user originally called the method on. This guarantee is necessary since there are applications of refinement that work on a per-object base: managing an object's resources via refinement, for example, would be meaningless if the refinement would not work on the original object of the user program but on a copy. This requirement therefore implies by-reference semantics or passing of addresses, coercion, and other features that avoid the introduction of copies. Of course, it is permitted to construct objects for helper classes.

The four subtasks, or constraints, just described essentially suffice to demonstrate the feasibility of controlling method semantics within a metaprogram at the user level. For its acceptance in practice, however, we additionally require the metaprogram to incur no run-time overhead. While it might be possible to implement the mutual dependency between base and derived class methods with a run-time callback mechanism, we request the call sequence be resolved at compile time. Moreover, the metaprogram should be usable with an acceptable amount of extra effort on part of the user.

5 The Requirements in C++

The previous section identified the requirements of a user program that implements refinement (and replacement) semantics. In summary, the program depends on a lazy binding facility and reference semantics, has to organize a callback and to accomplish inheritance tree traversal. To motivate the different design decisions we made in our C++ metaprogram, we now show how these requirements can be mapped to features in C++.

Lazy binding, to begin with, can be realized in C++ through the template mechanism. As the language specifies, template parameters are bound on demand, in particular is the code for a function template generated only when its template parameter is specialized. Using function templates we not only delay the compilation of a method but can also bind the template parameter in a way that brings the child method into play. In simplification, interface and body of a method look as follows:

```
template<class Inner>
void method() {  pre(); Inner(); post(); }
```

where `pre` and `post` are member functions that encapsulate the prefix and postfix of `method`. We note, for technical reasons, that template parameters of functions are type parameters. Whatever code we want to insert via parameter binding therefore has to be represented as a type; there exist standard techniques and idioms for how to do that.

Since the binding of template parameters is resolved at compile (instantiation) time, the required callback between the methods of the refinement relation and the method lookup have to be organized as a static callback. We therefore introduce a parameterized class that can hold all statically available information and the current type: the class `Rec<Self,Info>` is parameterized over the class types in which the member functions of the refinement relation are defined and an information class, which we discuss later.

The callback begins when a method passes on to `Rec` the type of the class it is member of (`Self`). Based on this type, and combined with the information from the second parameter, helper classes determine the next receiver and the next action to perform; the control returns when `method` is invoked on the child:

```
template<typename Info, typename Type>
struct Rec {
     void recur(Type& parent) {
           // look up the name of Type's child
           child& c = static_cast<child&>(parent);
           c.method<Info>();
     }
};
```

With the introduction of the Rec class, the previously given method body can be described more precisely: to initiate the static callback, method specializes Rec with the type of its owner class, Self; the second parameter, Info, is supported by the main program and passed on unchanged. The (still slightly simplified) body now reads as

```
template<class Info>
void method() {
    pre(); Rec<Self*,Info>(*this); post();
}
```

Three remarks are in place: first, the specialization with Self* instead of Self has technical reasons since C++ requires a complete type here while the definition of the class has not been finished yet. The use of a pointer type does not indicate any run-time indirection. Second, the different method invocations are called only on different slices of the very same object the user program supplies: method passes on the this object, and the Rec class downcasts it to call the next prefix on the child type; nowhere, however, is a new object created. Finally, all method bindings occur at compile time, thus avoid any run-time overhead.

The traversal of an inheritance tree, third, is easy to do at the object level given a language that has as many low-level features as C++; we have already seen the static_cast conversion operator. Since we are working within the template sublanguage, however, it will also be necessary to traverse the corresponding type tree and to trigger the right instantiations. Neither does C++ have direct support for such type tree traversal, however, nor is it possible for a the base class to provide a type alias to a derived, i.e., not-yet-existing class. We therefore resort to the standard technique of interface templates or traits [11] and introduce a childOf trait. The price to pay, of course, is that a user has to fill out this trait. Using the childOf trait, the Rec class can traverse the inheritance hierarchy downwards and, with the callback mechanism just described, enforce that the prefixes are executed in the correct order. The execution of the postfixes, in inverse order, is then for free. Once the bottom of the hierarchy has been reached and the recursion stops, the call stack in C++ guarantees that, starting with the innermost, leaf class, each method on the stack executes its prefix before it returns to its caller. When the call stack is empty, the outermost postfix has been invoked and the refinement is complete. The remaining question is how to control the depth of the refinement. This is done by the class Info that packages

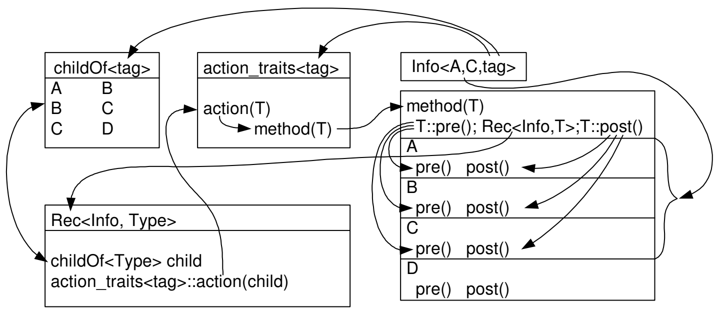


Fig. 2. Dependencies between the components of the framework: `method` instantiates `Rec`, which looks up the traits `childOf` and `action_traits`, to determine action and receiver type, then calls `method` on the child.

up the first and the last class in the refinement relation along with method to be refined.

```
struct Info<A,D,demo_tag>;     // refinement
struct Info<D,D,demo_tag>;     // replacement
```

6 Example

Most of the tasks explained in the previous section is transparent to the "end user." In demonstration, we take the class designer's view and illustrate a main program just from a normal user's angle.

Suppose we want to refine the method `demo`. From a user's point of view we can assume that the framework provides a class, say `RefineDemo`, that defines a `demo` (wrapper) function and determines that, in this particular refinement relation, the pre- and postfix are named `pre` and `post`. Given this framework class, the user derives the class `A` from it and defines the outermost action of the refinement as the body of `A::pre` and `A::post`, respectively. All other classes in the inheritance hierarchy, `B`, `C`, and `D`, are derived as usual; each of them defines the two methods `pre` and `post` with the same signature as `A::pre` and `A::post`. In addition, each class needs to specialize the `childOf` template, to help the metaprogram traversing the class hierarchy from the parent to the child. These steps done, the user has gained control over the method semantics.

Controlling the degree of refinement, including switching between refinement and replacement, is done by defining the "control type" `Info<Refinement,Root,Leaf>` and appropriately specializing its three parameters. In our example, there are two control types, `Info1` and `Info2`, both referring to the refinement relation represented by the `demo_tag`—the same used in the `childOf` definition—and both specifying `D` as the relative leaf. However, `Info1` lists class `A` as its relative root while `Info2` specifies class `D`. Binding `demo`'s type parameter to either `Info1` or `Info2`, the method `d.demo<Inner>` then either refines the pre- and postfixes in its base classes or replaces them.

```
#include "demo_lib.hpp"   // the framework
struct A : public RefineDemo {
  void pre (void) { std::cout << "A( "; }
  void post(void) { std::cout << ")A "; }
};
struct B: public A {
  void pre (void) { std::cout << "B[ "; }
  void post(void) { std::cout << "]B "; }
};
struct C: public B {
  void pre (void) { std::cout << "C{ "; }
  void post(void) { std::cout << "}C "; }
};
struct D: public C {
  void pre (void) { std::cout << "D< "; }
  void post(void) { std::cout << ">D "; }
};

template<> struct childOf<A*,demo_tag>   {  typedef B child; };
template<> struct childOf<B*,demo_tag>   {  typedef C child; };
template<> struct childOf<C*,demo_tag>   {  typedef D child; };

int main() {
  typedef Info<A,D,demo_tag> Info1;
  typedef Info<D,D,demo_tag> Info2;

  D d;
  std::cout << "   ----  Refinement ----\n";
  d.demo<Info1>();

  std::cout << "  \n---- Replacement ----\n";
  d.demo<Info2>();

  return 0;
}

[schupp@Strepponi]$ g++ user.C
[schupp@Strepponi]$ ./a.out
----  Refinement ----
A(   recurring down
B[   recurring down
C{   recurring down
D<   recursion stops
>D }C ]B )A
----  Replacement ----
D<   recursion stops
>D
```

Fig. 3. User-controlled method semantics: the instantiation of `Info` controls which methods are refined or whether refinement degenerates to replacement.

7 The Framework

In the methodology of object-oriented programming, the notion of a framework usually refers to a set of classes designed by advanced users for a special task and in a way that allows for easy extensions by application users. The framework in our case, however, implements a language feature rather than an application. It therefore makes sense to differentiate between the core metaprogram, which has to be written once, and the classes that are specific to a particular refinement relation. The code for the former is given in this paper, the code for the latter, though not difficult, typically should be provided by a library designer. In this section it also becomes apparent why we chose C++ as implementation language. The lookup of type names and of the various instances of the method to be refined, the recursive type binding and type tree traversal, the decision when the recursion stops, all this, as part of a (static) meta-program, can utilize C++, which implements these very steps within its template translation mechanism. Without the approach of a *meta*-program and the support by C++, the program size would be considerably larger.

7.1 The library designer

What distinguishes one refinement relation from another is not its operational semantics but merely the syntax: the names of the pre- and postfixes and of the refinement relation itself. The library designer's task, therefore, in essence is to introduce the (method) identifiers through which the metaprogram and the user program can communicate.

To continue the example of the previous section, we assume the task is to organize the refinement of the method `demo`. Altogether three methods have to be defined, then: first, the interface to the user program, the `demo` method itself, which extracts the relative root the user has provided and calls a helper function. Second, this helper function, `_demo`. This function is part of the static callback: it calls `pre` and `post` on a parameter that, in the course of the metaprogram, is bound to different types, and furthermore creates a `Rec` object that, as we will see in the next section, determines the child of `Self` and calls `_demo` on the child. The implicit (though documented) assumption is that all specializations of `Self` define methods `pre` and `post` and, conversely, that the names for the pre- and postfixes for other refinements do not cause name conflicts.

```
template<class Inner>
void demo() {
     typedef typename Inner::root root;
    _demo<Inner,root>(static_cast<root&>(*this));
}
template<class Inner, class Self>
void _demo(Self& s) {
     s.Self::pre(); Rec<Inner,Self*> r(s);  s.Self::post();
}
```

Both demo and _demo are encapsulated in a class, here RefineDemo, which serves as the base for the deriviation of user classes so that demo can be called on any such derivation. Except for the 4 identifiers involved, demo, _demo, pre, and post, the exactly same steps have to be taken for any other refinement.

The third and last method to define is the member action of a so-called action_trait. Again, the purpose is to communicate to the metaprogram information specific to the demo-refinement, here: which particular action to call in the static callback (see appendix A for the complete code).

```
template<typename InfoType>
void action(Type& t, InfoType&)  {
     t.template _demo<InfoType,Type>(t);
}
```

7.2 The metaprogram

The core of the metaprogram is the class Rec we have already seen (Sect. 5), which organizes the static combination of recursion and callback. Its body overloads the method identifier recur with an implementation for the base case, when the recursion stops, and one for the recursive case. In the recursive case, three steps suffice: given a type reference and the particular tag for a refinement, the name of this type's child, ch, can be retrieved from the childOf trait. This very tag and the child's type-name can be used to specialize and instantiate the action_trait, which then downcasts the parent object to a child object and invokes action on the child. The info object is passed along so that the subsequent instance of Rec can decide whether or not the recursion stops. For the complete code see appendix B.

```
void recur(false_t,Type& parent) {
   typedef typename childOf<Type*,typename Info::fun>::child ch;
   action_trait<child,typename Info::fun> next;

   Info info;
   next.action(static_cast<ch&>(parent),info);
}
void recur(true_t, Type& t) {}
```

8 Evaluation and Conclusion

With the implementation details and the example main program from the previous sections we can now evaluate the framework as a whole and conclude why a uniform, user-controlled view of replacement and refinement can be useful.

One of the major concerns in practice is whether the suggested framework, a program on top of the compiler's method dispatching, incurs any overhead in space and time. The technique of metaprogramming, however, ensures that all computation takes place at compile time. Moreover, all additionally created objects are either instances of empty classes, thus likely to be optimized away, or at least do not contain any data members. Also important from a practical standpoint, the user interface almost entirely hides the complexity of the metaprogram: except for the definition of the `Info` type, users merely exercise an object-oriented discipline. Lastly, since metaprograms can have surprisingly long compilation times, we point out that in our program the depth of the template nest is not large enough to cause any compilation problems. We should also mention that the code presented supports nullary functions only; to support methods of arbitrary arity the metaprogram needs to overload the wrapper method `action` for every arity.

Another question is whether there actually are situations where a user would like to have a choice between replacement and refinement and, furthermore, whether library designers would actually like to see their users given such choice. We want to emphatically answer both questions in the affirmative and believe that it is important that refinement and replacement can coexist not only in one inheritance hierarchy but also in one method. For an example one may consider the following algebraic hierarchy of multiplicative structures: an abstract class `Ring` and the two classes `VectorRing` and `ComplexVectorRing`, which clearly can be ordered so that each multiplication refines the previous one [10, 8]. What, however, if a user wants to extend this inheritance hierarchy by a matrix class? The specifics of matrix multiplication requires to break the refinement relation. If refinement and replacement coexist, users can simply switch to replacement semantics, thus are not hampered by earlier decisions of a library designer. None of the current object-oriented languages, however, gives programmers any means to specify the method semantics.

More flexible prove multimethods [3, 5], which provide optional refinement facilities such as `:after` and `:before` in CLOS. Yet, they too force to cast in stone what a method's semantics shall be and thereby exclude any diverging view a client might have. In particular is it impossible to associate more than one semantics with a method. If refinement and replacement can coexist in the same method, however, a method can both specialize its parent's method and be used independently from it. In the XML example in Sect. 3, for instance, it is easy to envision that a document design class can be used both as part of a larger, i.e., the company's design style and stand-alone. This flexibility is lost if the method designer needs to make an (irreversible) choice.

As often is the case, more freedom for a user implies more responsibility. We believe the advantages outweigh the risks here.

A The Library

```
#ifndef __DEMO_LIB_HPP
#define __DEMO_LIB_HPP
#include "refinement.hpp"

struct demo_tag{};
```

```
template<class Type>
struct action_trait<Type,demo_tag>
{
  template<typename InfoType>
  void action(Type& t, InfoType&)   // transfer control back to
  {                                 // RefineDemo::_demo
      t.template _demo<InfoType,Type>(t);
  }
};

struct RefineDemo {
  template<class Inner>             // user interface: extract the
  void demo() {                     // root, let _demo do the work
      _demo<Inner,typename Inner::root>
          (static_cast<typename Inner::root&>(*this));
  }

  template<class Inner,class Type> // static callback: transfer
  void _demo(Type& s) {            // control to Rec, schedule post
      s.pre(); Rec<Inner,Type*>r(s); s.post();
  }
};
#endif // DEMO_LIB_HPP
```

B The Metaprogram

```
#ifndef __REFINEMENT_HPP
#define __REFINEMENT_HPP

#include <iosfwd>
#include <string>
#ifndef TRACE
#define TRACE (1)
#endif

static int indent = 2;           // for demonstration purposes
void trace(const std::string& s, int indent) {
  if (!TRACE) return;
  while(indent-- && !(indent <0)) std::cerr << " ";
  std::cerr << s << std::endl;
}

// users control degree of refinement by specializing
// Root and Last
template<class Root, class Last, class FunTag>
```

```
struct Info
{
  typedef Root   root;
  typedef Last   last;
  typedef FunTag fun;
};

template<class Child, class FunTag>     // downwards traversal of
struct childOf;                         // the inheritance tree

template<class Type, class FunTag>      // encapsulation of
struct action_trait;                    // callback function


struct true_t{}; struct false_t {};
template<typename T1, typename T2>      // compile-time decidable
struct EQUAL                            // equality predicate
{
    static const bool RET = false;
    typedef false_t result;
};

template<typename T>
struct EQUAL<T, T>
{
    static const bool RET = true;
    typedef true_t result;
};

template<typename Info, typename Type>    // the metaprogram
struct Rec;

template<typename Info, typename Type>
struct Rec<Info,Type*>
{
  Rec(Type& parent) {
     // if Type == Info::last return, otherwise recur
     typedef typename EQUAL<Type,typename Info::last>::result res;
     recur(res(),parent);
  }
  void recur(false_t,Type& parent)      // recursive case
  {
     if (TRACE) trace("recurring down", indent);

     // look up the type-name of Type's child, create action trait
    typedef typename childOf<Type*,typename Info::fun>::child child;
    action_trait<child,typename Info::fun> next;

    // downcast the parent object and call action on the child
    Info   info;
    next.action(static_cast<child&>(parent),info);
  }
  void recur(true_t, Type& t) {         //  base case
    if (TRACE) trace("recursion stops", indent);
  }
};
#endif // __REFINEMENT_HPP
```

References

[1] K. Bruce. *Foundations of Object-Oriented Languages.* MIT Press, 2002.

[2] T. Budd. *Object-oriented Programming.* Addison-Wesley, 1997.

[3] C. Chambers. Object-oriented multi-methods in Cecil. In O. Lehrmann Madsen, editor, *Proc. of the 6th European Conf. on Object-Oriented Programming (ECOOP)*, volume 615, pages 33–56. Springer-Verlag, 1992.

[4] K. Czarnecki and U. Eisenecker. *Generative Programming—Towards a New Paradigm of Software Engineering.* Addison Wesley Longman, 2000.

[5] D. Bobrow et. al. Common Lisp Object System specification: X3J13 Document 88-002r. *ACM SIGPLAN Notices*, 1988.

[6] Mjolner Informatics. Introduction to Beta. Technical report, MIA 94-26, Nov 2000.

[7] S. Lippman. *C++ Gems.* Cambridge University Press, December 1996.

[8] O. Lehrmann Madsen, B. Moller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley, 1993.

[9] B. Magnusson. An overview of Simula. In J. Knudsen, M. Lofgren, O. Lehrmann-Madsen, and B. Magnusson, editors, *Object-oriented environments: the Mjolner approach*, pages 79–98. Prentice Hall, 1994.

[10] B. Meyer. *Object-oriented software construction.* Prentice Hall, 2nd edition, 1997.

[11] N. Myers. A new and useful template technique. In *C++ Gems* [7].

[12] K. Nygaard and O-J. Dahl. The development of the SIMULA language. In R. Wexelblat, editor, *History of Programming Languages*, pages 439–493. Academic Press, 1981.

[13] B. Stroustrup. *The C++ programming language, 2nd. Ed.* Addison-Wesley, 1994.

[14] W3C. Extensible markup language (XML) 1.0 (second edition). Technical report, http://www.w3.org/TR/2000/REC-xml-20001006, Oct 2000.

[15] P. Wegner. Dimensions of object-oriented language design. In *Proc. of OOPSLA'97*, pages 168–182, 1997.

Java-Style Variable Binding in C++

Thomas Becker

Zephyr Associates, Inc.
Zephyr Cove, NV `thomas@styleadvisor.com`

Abstract. This paper discusses how Java's variable-to-object binding can be emulated in C++. An example from commercial software development is described where this emulation was a factor in bringing the software to market in time and making it commercially successful.

1 Introduction

Ever since the C++ language saw the light of day, it has had competition from higher-level, more run-time intensive object-oriented languages. In the early nineteen-nineties, it was C++ vs. Smalltalk and CLOS, followed by C++ vs. Java, and, most recently, C++ vs. C♯. This debate takes place on all levels, in the academic world as well as in company meetings where decisions have to be made about which language to use for a particular software project or even for an enterprise-wide IT strategy. Needless to say, this is an overwhelmingly complex issue that cannot be fully addressed in a paper like this. However, there is one aspect of this discussion that I believe has been consistently underemphasized and overlooked, namely, the possibility of emulating idioms and paradigms from one language in another. Using the variable X for the name of the high-level object-oriented language du jour, my contention in the C++ vs. X discussion has always been, "You can emulate X in C++, but not vice versa." My main source of proof for this claim is James Coplien's 1992 classic "Advanced C++ Programming Styles and Idioms" ([1]). Large parts of this book are dedicated to the development of techniques to emulate idioms from Smalltalk in C++. Among other things, Coplien shows how the loose variable-to-object binding, the run-time type system, and the automatic memory management of Smalltalk can be emulated in C++. Reading Coplien's book today, it is quite clear that similar techniques can be used to achieve the same emulation for many idioms from Java or C♯.

It has long been my impression that Coplien's work and the numerous possibilities that it hints at have not received the kind of attention that they deserve, certainly not among practicing software engineers. Although many of his ideas have been taken up and developed further in the patterns discussion, the fact that he was primarily interested in emulating Smalltalk and CLOS idioms in C++ has received precious little attention. The current interest in multi-paradigm programming with C++ tends to focus on the emulation of non-object-oriented paradigms such as functional programming in C++. I strongly believe that it is important to broaden this discussion a bit and also investigate the emulation in C++ of facets and idioms from other object-oriented languages, especially high-level ones such as Java. When it comes to making a decision between C++ and, say, Java, people are often unaware of the fact that to quite an extent, you can have your cake and eat it. Using C++, it is possible to embed Java idioms "locally" in your software architecture, wherever appropriate, thus retaining the best of both worlds. This paper shows how Java's variable-to-object binding can be emulated in C++, and it reports on an example from commercial software development where this emulation was not only successful, but actually critical in bringing the product to market in a timely fashion and subsequently making it commercially successful.

2 The Example: Financial Analytics Software

The example on which this paper reports arose during the early design phases of a financial analytics program. This program is a rather complex desktop application that consists of a number of different components performing such diverse tasks as chart display, database access, number crunching, and the like. As it happens so often in the software industry, the team was expected to achieve both record time to market and long-term maintainability. Also, in view of the resources available, it was to be expected that some of the work would have to be delegated to outside contractors. Given these constraints, it was absolutely critical to base the architecture on strong modularization. Every effort was made to design the different components almost like libraries that expose well-defined, lean interfaces but have little or no knowledge of the context that they are being used in[1].

However, there was one particular kind of object that was going to blatantly violate this principle of strong modularization, namely, the portfolio object. The general purpose of the software is to analyze the historical return data of portfolios, and therefore, the one object that lies at the heart of the architecture is the portfolio object. This is a rather large and memory-intensive object that contains both the numbers that constitute the historical return data of the portfolio and all manner of qualitative data. `Portfolio` objects have to be passed around frequently and in large numbers throughout the entire software, with little or no respect for module boundaries.

Experiments performed with a simple prototype, combined with some back-of-the-envelope calculations, produced conclusive evidence that because of the size and complexity of portfolio objects, passing them around by value was way too costly in general, both in terms of time and space. The rule was going to have to be, do not make value copies of portfolio objects. This came as quite an irritation, for at least three reasons:

[1] Good software design pays off in unexpected ways, too: the strongly modularized architecture was later instrumental in the effortless reuse of components from the desktop product in the server backend of a related Web product.

1. The only way to enforce this rule was to admonish everybody to abide by it. This meant that the rule was going to create an additional headache for every single developer on the project, and it was going to be violated anyway.
2. In C++, passing objects by reference throughout a large piece of software creates lifetime issues that are difficult to track and often end up being the source of hard-to-find bugs.
3. Given the fact that C++ has an official way to cast away constness, the const qualifier on a `const` reference can hardly be considered more than a hint. Therefore, passing portfolio objects by reference threw open the gates for anybody to modify a globally shared object from anywhere in the code. The crux here is that in C++, even if a class is read-only in the sense that it does not expose any member functions that modify the object's state, anybody who holds a reference to an object can still modify it by assigning to it. That is because in C++, assigning to a reference has value semantics: it assigns the value of the right hand side to the object referenced by the left hand side.

While none of the three items above entirely precluded the use of C++ references for passing around portfolio objects, together they were serious enough to earn the issue a good spot on the list of project risks. At the time, however, this list was in dire need of having items removed from it rather than added to it. Fortunately, it is possible to make the problem go away by emulating Java-style variable binding for objects of this particular class.

3 Variable Binding: Java vs. C++

As far as variable binding is concerned, the difference between C++ and Java is perhaps best understood by looking at function argument passing. In C++, function arguments can be passed by value or by reference. Passing by value holds no surprises and is in fact irrelevant for the problem at hand. Now let us assume that `X` is a class that has a public default constructor, copy constructor, and assignment operator, and a public member function `memfoo` that modifies the state of the `*this`-object. Now consider the following function:

```
void foo(X& refX)
{
  refX.memfoo();
  X anotherX;
  refX = anotherX;
}
```

and suppose that `foo` gets called like this:

```
X anX;
foo(anX);
```

When the object `anX` is passed to the function `foo`, the local reference variable `refX` is initialized to refer to the caller's object `anX`, just as if we had written

```
X& refX = anX;
```

The line

```
refX.memfoo();
```

in the definition of `foo` will modify the caller's object `anX`, because that's what `refX` refers to. The interesting part is

```
X anotherX;
refX = anotherX;
```

It always surprises me how many seasoned C++ professionals are hesitant or unsure when asked what these two lines do. The fact of the matter is that the assignment in the second line has value semantics. The assignment operator of X is called with a left hand side of `anX` and a right hand side of `anotherX`. The reference variable `refX` is not reset to refer to `anotherX`. In fact, there is no way to achieve that. C++ references can only be initialized, but not redirected. For references, copy construction and assignment have different semantics. I have heard people call this behavior unintuitive, irritating, even wrong. Personally, I am not passionate either way. But I will admit that I find it a bit surprising that the C++ literature rarely emphasizes this point and, to the best of my knowledge, makes no attempt to explain the rationale behind this design decision.

Be all that as it may, the important point for our problem is that Java behaves differently in this respect. First of all, there is only one way to pass function arguments in Java. The choice between passing by value and passing by reference is not offered. Therefore, to turn the example above into valid Java code, we have to modify the function `foo` a little:

```
void foo(X argX)
{
  argX.memfoo();
  X anotherX = new X();
  argX = anotherX;
}
```

Many Java developers, when asked whether function arguments in Java are passed by value or by reference, will answer, "by reference." That's not too bad an answer, but interestingly, James Gosling himself begs to differ. In his introductory Java book, Gosling says ([2], p. 40): "Some people will say incorrectly that objects in Java are 'pass by reference.' [...] Java does not pass objects by reference; it passes object references by value. [...] There is exactly one parameter passing mode in Java—pass by value—and that helps keep things simple."

The little example above illustrates what that means: in Java, the function `foo` and its argument `argX` behave much like they did before in C++. The one difference is in the two lines

```
X anotherX = new X();
argX = anotherX;
```

The assignment

```
argX = anotherX;
```

causes `argX` to be reset to refer to the object `anotherX`. The caller's object `anX` is completely unaffected by this assignment. Actually, the object `anX` can no longer be accessed at all in the body of `foo` after `argX` has been assigned to. As a matter of fact, all variables in Java, not just function parameters, behave like this: they are references that get reset when assigned to. Hence James Gosling's statement about references that are passed by value.

James Coplien has described this kind of "loose" variable-to-object binding by saying, "Binding a variable to an object is like sticking a label on it" ([1], p. 134). He was really referring to variable binding in Smalltalk, where, in addition to the reference-like nature of variables, there is also little or no compile-time type information associated with a variable. Still, his metaphor works quite well for Java, too. The difference is that as far as the type model is concerned, Java behaves pretty much like C++. Therefore, we would have to say that binding a variable to an object in Java is like sticking a label on it, but each label carries information about what kind of object it can go on. In other words, as far as variable binding is concerned, Java is somewhere in the middle between C++ and Smalltalk, combining the loose variable-to-object binding of Smalltalk with the strong compile-time typing of C++. In view of this situation, it should come as no surprise that James Coplien's techniques for emulating Smalltalk-style variable binding in C++ ([1], Chapter 5) can easily be adapted to do the same for Java-style variable binding. Before I explain this emulation, I will discuss why it solved our problem with the portfolio class so well.

4 The Solution: Emulating Java-Style Variable Binding in C++

Suppose for a moment that we have magically made our portfolio class behave as if it were a Java class. Then the three problems listed above disappear. Problem 3 is gone by definition: it is no longer possible to modify a referenced object by assigning to it. Since our portfolio class is read-only in the sense that it does not expose any non-const methods beyond construction, it is thus not possible at all for anybody to modify a referenced object. Problem 2, concerning the lifetime issues of referenced objects, will also disappear because, as we'll see shortly, automatic lifetime management via reference-counting comes naturally with the emulation of Java-style variable binding. Finally, Problem 1 is gone because there isn't a rule anymore that anyone should have to abide by. Naturally, our developers wanted to know, and deserved to know, how portfolio variables and objects behaved. But the answer was ridiculously easy: "Think Java." That's all there was to it, and it wasn't even a simplification[2]. In a situation where not everybody is a Java buff, one may have to refer to a different language, or even explain the whole thing on a more conceptual level. Either way, the point is that the developers who use the class in question do not come away with the impression that there is some bizarre C++ hack at work that they have to beware of. Instead, variables and objects behave according to an easily stated and well-understood idiom that is being emulated by the implementation of the class in a way that is transparent to the client.

5 The Implementation

It turns out that emulating Java-style variable binding in C++ hardly requires the invention of groundbreaking new techniques. It is, as we will see shortly, closely related to several well-known and well-understood idioms. However, since it is not identical to any one of these idioms, I will refer to it by its own acronym "JSVB," for "Java-Style Variable Binding." The bare essence of the implementation is shown below. For readability, all method implementations are in the class declarations. In reality, the code is divided into headers and implementation files in such a way that clients of the `Portfolio` class will see only a forward declaration of the `PortfolioBody` class.

```
class Body
{
public:
  Body() : m_nRefCount(1) {};

  virtual ~Body()
  { assert(0 == m_nRefCount); }

  void AddRef()
  { AtomicIncrement(m_nRefCount); }

  void Release()
  {
    // Looks too good to be true but is thread-safe
    if( 0 == AtomicDecrementAndTest(m_nRefCount) )
```

[2] This was of course partly because all this happened during the heyday of the Java hype, when the Java One conference in San Francisco sold out faster than a Radiohead concert. Moreover, every C++ developer at the time harbored some secret doubts if he or she was missing the boat by writing C++ rather than Java. Therefore, saying that we were emulating Java here was a subtle psychological trick (unintended, of course) to boost project morale.

```
      delete this;
  }

private:
  volatile int m_nRefCount;
};

class PortfolioBody : public Body
{
public:
  void Load(string strDatabaseID)
  { /* Load from database */ }

  // Would compile as non-const. Must rely on coding discipline
  // to make this const because Portfolio::Foo() is const.
  int Foo(int) const;
};

class Portfolio
{
public:
  Portfolio() : m_pBody(NULL){}

  void Load(string strDatabaseID)
  {
    if( m_pBody )
      throw DatabaseException("Object not null");
    m_pBody = new PortfolioBody;
    m_pBody->Load(strDatabaseID);
  }

  Portfolio(const Portfolio& rhs)
  {
    m_pBody = rhs.m_pBody;
    if( m_pBody )
      m_pBody->AddRef();
  }

  // this'll end up being virtual...
  ~Portfolio()
  { if( m_pBody ) m_pBody->Release(); }

  Portfolio& operator=(const Portfolio& rhs)
  {
    if( rhs.m_pBody == m_pBody )
```

```
      return *this;
    if( m_pBody )
      m_pBody->Release();
    m_pBody = rhs.m_pBody;
    if( m_pBody )
      m_pBody->AddRef();
    return *this;
  }

  // Interface methods are forwarded to the body
  int Foo(int i) const
  { assert(m_pBody); return m_pBody->Foo(i); }

private:
  PortfolioBody* m_pBody;
};
```

The portfolio class itself holds as its only data member a pointer to an implementation object. All public member functions, with the exception of construction, destruction, and assignment, are forwarded to the implementation object. So far, our JSVB resembles the well-known pimpl (pointer to implementation) idiom, also known as the compiler firewall idiom. This idiom is discussed in great detail in Items 26–30 of Herb Sutter's first book ([3]). Our body class corresponds to the implementation class in the pimpl idiom. There are two differences. Firstly, in the pimpl idiom, only the private portion of the class itself is delegated to the implementation helper class. In our case, the class delegates its implementation and forwards its entire interface with the exception of construction, destruction, and assignment to the body class. Secondly, in the pure pimpl idiom, each object of the class has its own implementation object. In our case, the body has a reference count and is shared when an assignment or copy construction takes place. This sharing of body objects makes our implementation reminiscent of the time-honored COW (copy-on-write) idiom, which is discussed e.g. in Items 13–16 of Herb Sutter's second book ([4]), and also by James Coplien in [1], where it is called the handle-body idiom. Our `Portfolio` class corresponds to the handle, and our `PortfolioBody` is the body of the handle-body idiom. The difference between the handle-body, or COW, idiom and our JSVB is that in the handle-body idiom, the sharing of bodies is merely an optimization that remains transparent to the client. As soon as the state of the body is modified through a handle, that handle ceases to share the body and obtains its own copy instead. In JSVB, there is no such copying. If the state of the body is modified through a handle, all handles that share this body will see the change. That's works-as-designed in Java.

Finally, it should be mentioned that our JSVB also bears a strong resemblance to legions of reference-counted smart pointer implementations. The crucial difference here is that our `Portfolio` class passes itself off as the actual object, thus saving clients from having to use pointer syntax when working with

`Portfolio` variables. Another way to express this would be to say that our `Portfolio` is not a smart pointer, but a smart reference.

Of course it is true that our problem could also have been solved with a suitable smart pointer to `PortfolioBody` objects. A similar statement can be made about dumb pointers and C++-style references: everything that you do with a C++-style reference could have been achieved with a dumb pointer. But references were added to C++ for a reason: they are often much more intuitive than the corresponding pointer solution. The same is true here. From a pointer, smart or dumb, we expect that we can get a C++-style reference to the pointee via `operator*()`. This would have to be suppressed in our case. Java-style smart references solve the problem in a much more intuitive way. By using JSVB instead of a specially designed smart pointer, we have lifted the solution from a technical to a conceptual level.

6 JSVB and Concurrency

The issue of thread-safety and JSVB deserves some comment. The fact that Java-style variables always and unconditionally behave like references must of course be taken into account when it comes to protecting objects from concurrent access. The JSVB implementation is only responsible for making the variable binding itself thread-safe. For example, consider the following four lines of code:

```
Portfolio p1;
p1.Load("MSE.FMAGX");
Portfolio p2 = p1;
Portfolio p3 = p1;
```

Now suppose that the third and fourth lines are executed concurrently in two different threads. The JSVB implementation is responsible for making this safe, i.e., the concurrent incrementing of the reference count must be performed correctly. The use of atomic increment and decrement in the implementation of the body class ensures that. (It is perhaps a bit surprising at first that this can be achieved with atomic integer operations alone, without the use of a lock. The body's destructor, in particular, may look suspicious to you at first glance. See e.g. Item 16, Part 4, of Herb Sutter's second book ([4]) for an explanation of thread-safety using atomic integer operations only.) If, on the other hand, a client writes code like this:

```
Portfolio p1;
p1.Load("MSE.FMAGX");
Portfolio p2;
Portfolio p3;
p3 = p1;
p3 = p2;
```

and the last two lines are executed concurrently without the use of a lock, then a null pointer may get dereferenced. It is the client's responsibility to prevent that. That's completely intuitive to a C++-programmer: assigning to the same variable concurrently is not per se safe. Now suppose `FooMod` were a `Portfolio` member function that modified the state of the portfolio body, and consider the following code:

```
Portfolio p1;
p1.Load("MSE.FMAGX");
Portfolio p2 = p1;
p1.FooMod();
p2.FooMod();
```

When the last two lines are executed concurrently, it is the client's responsibility to protect the object from being corrupted. From a C++ point of view, this is surprising, because under C++-style variable binding, `p1` and `p2` would be two entirely different objects, and there wouldn't be a concurrency issue (unless, of course, `FooMod` is a function that modifies static or global data). The price one has to pay for Java-style variable binding is that in a situation like this, there is a concurrency issue. In our case, this was a moot point because portfolio objects represent data from persistent storage and are not subject to modification by the program.

7 Additional Issues

The analogy with Java ends when it comes to constructing `Portfolio` objects. The default constructor will create a null reference, as it would in Java. In order to actually create an object, however, clients must call the member function `Load` to read a portfolio from persistent storage. To enforce the read-only nature of `Portfolio` objects, `Load` throws an exception when called on a non-null object.

Java doesn't have the notion of a `const` variable or a `const` member function. However, when a C++ class emulates Java-style variable binding, as is the case with our `Portfolio` class, then variables of that type can and will be declared as `const` in the C++ code. Since there is no Java-style behavior to emulate when it comes to constness, the only guideline is that `const` variables with Java-style binding should behave the way a C++ programmer would expect them to behave:

1. A `const` variable can only be initialized, but not assigned to.
2. Only `const` methods can be called on a `const` object.
3. `const` methods do not change the state of an object.

Requirements 1 and 2 are satisfied by our JSVB implementation without further effort. For requirement 3 to be satisfied, it is necessary to insure that all `const` methods of the handle class forward to `const` methods of the body. Unfortunately, there is nothing in C++ that would allow us to enforce that. The body pointer of a `const` handle object is a `const` pointer, but not a pointer to `const`. It would certainly be nice to have C++ language support for deep constness, just as it would be nice to have support for forwarding of member functions to implementation objects. Until then, we're on our own with nothing to rely on other than coding discipline.

Another issue that came up with our JSVB implementation was inheritance. Needless to say, the portfolio ended up being modeled not by a single class, but by a class hierarchy. Since our portfolios are always loaded from persistent storage, and the actual type of the portfolio that is being loaded is not known beforehand, the notion of virtual constructors also came into play. With our `Portfolio`, we were able to deal with all these issues without ever having to compromise the Java-style variable binding. This is not the place to go into further detail, but I'll have to say that our solution did feel somewhat ad hoc in several respects. The question of how to best deal with inheritance and virtual construction in the presence of JSVB certainly deserves some more investigation.

8 Conclusion

Java-style variable binding can be emulated for C++ classes and class hierarchies. Automatic memory management comes naturally with this emulation. This kind of variable-to-object binding is useful for large read-only objects. Such objects can then be passed around as Java-style references. This takes care of time and space efficiency concerns, and it definitively prevents clients from modifying referenced objects through assignment. For large objects that are not read-only, Java-style variable binding can still be useful because it provides automated object lifetime management without requiring clients to use pointer syntax, as would be the case with smart pointers.

References

[1] Coplien, J.: Advanced C++ Programming Styles and Idioms. Addison-Wesley (1992)

[2] Arnold, K. and Gosling, J.: The Java Programming Language. 2nd edn. Addison-Wesley (1997)

[3] Sutter, H.: Exceptional C++. Addison-Wesley (1999)

[4] Sutter, H.: More Exceptional C++. Addison-Wesley (2001)

Dynamic Inheritance for a Prototype-based Language

Koji Kagawa

RISE, Faculty of Engineering
Kagawa University
2217-20 Hayashi-cho, Takamatsu, Kagawa 761-0396
JAPAN
kagawa@eng.kagawa-u.ac.jp

Abstract. ECMAScript (a.k.a. JavaScript) is a *prototype-based* object-oriented language. Unlike many other object-oriented languages, it allows programmers to add new operations to existing objects. Moreover, it has anonymous functions (i.e. λ-expressions). Due to the combination of these features, it already supports the functional programming paradigm. With a small extension, it can also support other paradigms such as lazy functional and logic programming paradigms by facilitating introduction of data structures peculiar to these programming paradigms. This paper proposes such an extension and shows some examples.

1 Introduction

ECMAScript [1], also known as JavaScript, is widely used in order to add dynamic behaviours to WWW pages. However, its interesting features as an object-oriented language are probably less well-known. Among them, *prototype-based* inheritance is one of the most interesting. Unlike class-based object-oriented languages, each object constructor of ECMAScript has a special property (field) called `prototype`. If an object created from this constructor is sent a message and the object itself does not have a corresponding property (*i.e.* a method or a field), the message is directed to the constructor's prototype object. Moreover, we can freely add new properties to objects including prototype objects on the fly. In other languages, it is often difficult to add both new variants and new operations to an existing set of variants and operations without changing source code. In functional languages, it is easy to add new operations to an existing datatype while adding a new variant implies modifying many related functions. On the other hand, in most object-oriented languages, it is easy to add a new variant as a class while adding new operations requires modifying the classes which constitute the datatype. This problem — *the extensibility problem* (*e.g.* [4, 18]) — does not exist in ECMAScript[1]. This means that ECMAScript can be readily used as an untyped strict functional language.

[1] Of course, ECMAScript is an untyped (or a dynamically-typed) language and cannot be compared on the same level with strongly-typed languages such as Java.

As an example, we will define constructors for lists and operations on them — a datatype popular in functional languages but less frequently used in object-oriented languages.

In ECMAScript, constructors are just functions which initialize properties of the implicit argument `this`.

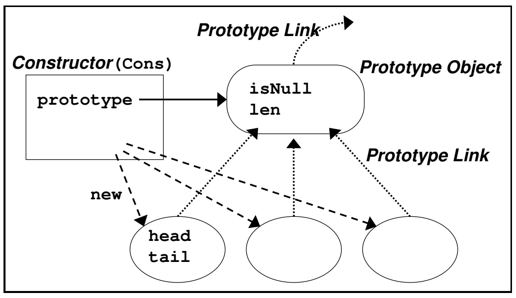
```
function Cons(x, xs) {
  this.head = x;
  this.tail = xs;
}
function cons(x, xs) { return new Cons(x, xs); }

function Nil() { /* do nothing */ }
var nil = new Nil();
```

The keyword `function` introduces function definitions. Here, `Cons` and `Nil` are *constructor* functions. By the `new` operator, a new object is allocated and passed to the constructor function. For example, the expression "`new Cons(x, xs)`" creates a new object and passes it as `this` to the `Cons` function. This expression, as a whole, returns the object initialized by `Cons`.

By default, a constructor's `prototype` property is initialized to a fresh object of type `Object` — the super type of any objects which contains a minimal set of methods such as `toString`. Then, we can add methods to the prototype object. For example, the following code adds `isNull` and `len` methods to `Cons` and `Nil`.

```
Nil.prototype.isNull = true;
Cons.prototype.isNull = false;

Cons.prototype.len = function () {
  return 1+this.tail.len();
}
Nil.prototype.len = function () {
  return 0;
}
```

Expressions introduced by the keyword `function` without an identifier such as "`function (...) {...}`" are anonymous functions in ECMAScript. They correspond to "`(lambda (...) ...)`" in Scheme and "`fn ... ⇒ ...`" in Standard ML. The relation among objects is illustrated by the following diagram.



Then, “`cons(1, cons(2, nil)).len()`” evaluates to 2. Thus, though ECMAScript does not have pattern matching, thanks to prototype-based inheritance, we can write programs in a way very close to those which use pattern matching. We do not need to rely on *visitor patterns* [5] to emulate algebraic data types and to add new operations to them later.

SpiderMonkey, an implementation of ECMAScript by Netscape, has another interesting feature, though it is not in the ECMAScript standard. It provides getter/setter mechanism for defining properties. That is, we can associate functions to a field (instance variable) definition so that these functions are called when the field is accessed. For example[2],

```
function Foo (x) { this.hidden = x; }
Foo.prototype.x␣getter = function () {
  return this.hidden*2;
}
Foo.prototype.x␣setter = function (y) {
  this.hidden = Math.floor(y/2);
}

function foo(x) { return new Foo(x); }
var a = foo(0);
a.x=9;
print(a.x);
```

In the second last line, the statement “`a.x=9`” is evaluated as if “`a.x␣setter(9)`,” and “`print(a.x)`” is evaluated as if “`print(a.x␣getter())`” and outputs 8. In this example, `a.x` always evaluates to an even number. Thus, getter/setter mechanism allows properties to be calculated dynamically.

Currently, SpiderMonkey does not seem to allow getter/setter definition for the special property `prototype`. However, if the `prototype` property can be computed dynamically, an interesting programming technique becomes possible. We will show that we can define wrappers to simulate lazy data structures and logical variables — data structures peculiar to lazy functional and logic programming languages respectively. By using those wrappers, most existing functions for the base datatype (*e.g.* list) can be reused, as they are, for the derived datatype (*e.g.* lazy list). Instead, if we were forced to rewrite library functions for the derived datatype, or to insert coercions from the derived type to the base type in all necessary places, it would be a very painful task for programmers.

SELF [15] is also a prototype-based language. The idea of “dynamic inheritance” that the prototype object can be changed dynamically was used in [14][3] in order to implement “multiple behaviour modes” in SELF. There, for example,

[2] JavaScript 2.0 (`http://www.mozilla.org/js/language/js20/`) proposal uses the syntax “`get␣`*name*” instead of “*name*`␣getter`.” The syntax “*name*`␣getter`” is used in the current version (1.5 pre-release 4a) of SpiderMonkey.

[3] The author would like to thank an anonymous referee for suggesting a reference.

the prototype of a polygon object is changed between a *boxed* polygon and a *smooth* polygon. Then the appearance of a polygon on the screen changes.

In the following in this paper, dynamic inheritance is used in order to implement more abstract wrappers. Prototypes can be even *created* on the fly. In some respect, the proposal resembles *mixins* [2, 4] since it can extend *any* existing datatype. However, the intention of our wrappers is to change the behaviour of existing methods slightly, while mixins usually add new methods to existing classes. In this sense, our proposal can be called *decorator mixins* since it implements a generic decorator pattern.

The rest of the paper is organized as follows. Section 2 introduces dynamic prototype into ECMAScript and defines a wrapper for lazy data structures (delayed computation). Section 3 shows some examples. Section 4 gives a definition of wrappers for logical variables. Section 5 concludes and gives future directions.

2 Dynamic Prototype

We define a constructor for the "delay" wrapper as follows.

```
function Delay(f) {
  this.memo = null;
  this.f    = f;
}

function delay(f) { return new Delay(f); }

function force(x) {
  if (x.memo==null) { x.memo = x.f(); }
  return x.memo;
}
```

This is a standard definition found in many strict functional languages such as Scheme [9] and ML [12, 10]. It uses the field `memo` to avoid repeated invocation of the delayed computation `f`. (This is a simplified version — actually, further care must be taken when the delayed computation refers to its own result or when it throws an exception.) By wrapping lists with this object, we can obtain lazy lists. For example, we can define an infinite list of numbers $[n,\ n+1,\ \ldots]$ as follows:

```
function from(n) {
  return delay(function () { return cons(n, from(n+1)); });
}
```

(Note that this definition is in the *even* style in the terminology of [16]. On the other hand, the odd style definition is as follows:

```
// function from(n) {
//  return cons(n, delay(function () { return from(n+1)); });
// }
```

In general, the even style definitions delay more computation and are easier to use.)

Lazy data structures are a very useful programming technique especially in functional programming languages. In [7], it is even claimed as "the most powerful glue functional programmers possess." Therefore, it is important to be able to treat lazy data structures as first-class citizens. In practice, however, lazy data structures are not used so frequently in strict programming languages. This is probably because we cannot reuse existing rich library functions for the standard lists and therefore we must redefine huge amount of library functions for such a variation of list. In prototype-based languages such as ECMAScript, however, where we can add necessary methods to existing datatypes later, we can avoid such repeated definition of essentially the same functions.

To be more specific, we can make methods such as `head` and `tail` directly applicable to delay objects so that expressions such as "`from(1).head`" and "`from(1).tail.tail.head`" are possible. Of course, this is made possible by defining `head` and `tail` methods for `Delay.prototype`.

```
// Delay.prototype.head getter = function () {
//  return force(this).head;
// }
// Delay.prototype.tail getter = function () {
//   return force(this).tail;
// }
```

However, this is somewhat strange and inconvenient — if `delay` is used to wrap objects of another type such as Tree, further methods must be added for `Delay`. Since `Delay` itself is generic, it breaks modularity. Moreover, the number of the properties to be defined is much larger in practice, and it becomes cumbersome to redefine all the properties. If we can compute the prototype object dynamically, such tedious tasks can be avoided completely.

As a first try, we might like to write:

```
// Delay.prototype getter = function () {
//   return force(this);
// }
```

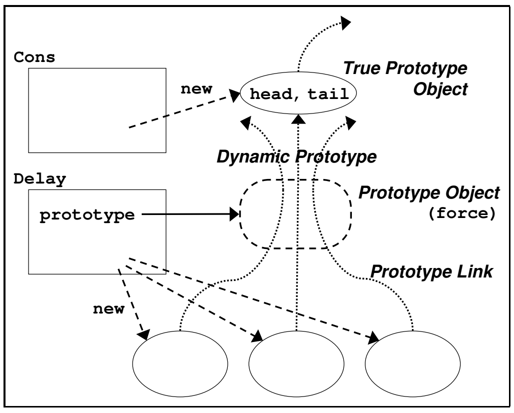
The intention here is as follows. When an object does not have a property in itself and we need the `prototype` property of `Delay`, the above getter function for `prototype` is called with `this` bound to the object which is initially sent a message.

Unfortunately, there is a problem here. In the ordinary semantics of ECMAScript, `this` above is bound to the constructor function `Delay` itself and not to the object initialized by `Delay`.

In this paper, we propose the following extension — if the constructor's `prototype` property is a function object which takes one parameter, it is called with the argument representing `this` object and its return value is used as `prototype` when looking up a property. We will call this feature *dynamic prototype*. Then, we can replace the above definition by:

```
  Delay.prototype = force;
```

The situation is illustrated by the following diagram.



When "`from(1).head`" is evaluated, "`from(1).f ()`" is computed via `force` and the `head` property of the result of this call is returned. It is worth emphasizing here that library functions which are defined for non-lazy list, for example,

```
  function take(xs, n) {
    if (n==0) {
      return nil;
    } else {
      return cons(xs.head, take(xs.tail, n-1));
    }
  }

  function listToString(xs) {
    var buf = "[", str = "";
    for (; !xs.isNull; xs=xs.tail) {
      buf += (str+xs.head);
      str = ", ";
    }
    return buf+"]";
  }
```

can be reused, without any modification, to lazy lists. Without dynamic prototype, such library functions must be all rewritten, which discourages programmers to use lazy data structures.

3 Examples

In this section, we will show how the sieve of Eratosthenes — a very standard programming example for lazy functional languages — is defined in our extended ECMAScript.

First, we define standard library functions for lazy lists as follows:

```
function map(f, xs) {
  return delay(function () {
    if (xs.isNull) {
      return nil;
    } else {
      return cons(f(xs.head), map(f, xs.tail));
    }
  });
}

function filter(p, xs) {
  return delay(function() {
    if (xs.isNull) return xs;
    else if (p(xs.head)) {
      return cons(xs.head, filter(p, xs.tail));
    } else {
      return filter(p, xs.tail);
    }
  });
}

function iterate(a, f) {
  return delay(function() {
    return cons(a, iterate(f(a), f));
  });
}
```

The expression `map(f, xs)` applies function `f` to all the elements in `xs` and returns the list which contains the results, `filter(p, xs)` returns the list of elements of `xs` which satisfies the predicate `p`, and `iterate(a, f)` returns the infinite list `[a, f(a), f(f(a)), f(f(f(a))), ...]`.

The list of prime numbers can be defined as follows:

```
function sieve(xs) {
  return filter(function (x) {
    return x % xs.head != 0;
  }, xs);
}

var primes = map(function(x) { return x.head; },
                 iterate(from(2), sieve));
```

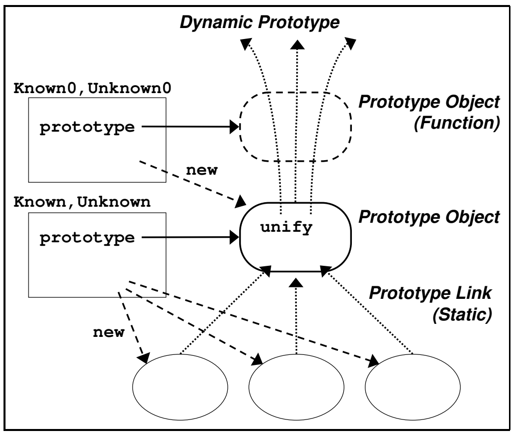
The expression `primes` itself is an infinite list, and "`listToString(take(primes, 10))`" evaluates to `"[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]"`. Note again that `listToString` and `take` are defined with strict lists in mind.

4 Logical Variables

We can also define wrapper constructors for logical variables — variables which may be instantiated to other objects or unified with other logical variables like variables in Prolog.

```
function Unknown0() { /* auxiliary constructor */ }
Unknown0.prototype = function (self) { return self.link; }

function Unknown() { this.link = null; }
function unknown() { return new Unknown(); }
Unknown.prototype  = new Unknown0 ();

Unknown.prototype.bind = function (y) {
  if (this.link==null) {
    this.link = y;
    return true;
  } else return false;
}

function Known0() { /* auxiliary constructor */ }
Known0.prototype = function (self) { return self.value; }

function Known(v) { this.value = v; }
function known(v) { return new Known(v); }
Known.prototype  = new Known0 ();
```

In this example, since some methods must be defined for `Known` and `Unknown`, we need auxiliary constructors `Known0` and `Unknown0`. There is a static prototype before accessing a dynamic prototype. This is illustrated by the following diagram.



Though the two constructors `Known` and `Unknown` are similar at a glance, the `link` field of `Unknown` is mutable and is bound to objects of `Unknown` or `Known` while the `value` field of `Known` is immutable and is initialized to objects of other types. We define `unify` method for `Known` and `Unknown` as follows.

```
Unknown.prototype.deref = function () {
  if (this.link==null) return this;
  else return this.link.deref;
}
Known.prototype.deref = function () { return this; }

Known.prototype.unify = function(y) {
  return y.unifyAux(this);
}
Unknown.prototype.unify = function(y) {
  if (this===y) return true;
  if (this.link == null) return this.bind(y.deref());
  return this.link.unify(y);
}

Known.prototype.unifyAux = function(x) {
  return x.value.unifyStructure(this.value);
}
Unknown.prototype.unifyAux = function(x) {
  if (this.link==null) return this.bind(x);
  else return this.link.unifyAux(x);
}
```

We assume that `unifyStructure` method is defined for the constructors involved in unification which recursively invokes `unify` for their each field. The definitions of `unifyStructure` for `Cons` and `Nil` are as follows:

```
Object.prototype.unifyStructure = function(y) {
  return this==y;  /* default unifyStructure */
}

Cons.prototype.unifyStructure = function(y) {
   return y.unifyCons(this.head, this.tail);
}
Nil.prototype.unifyStructure = function(y) {
  return y.isNull();
}

Cons.prototype.unifyCons = function(x, xs) {
  return x.unify(this.head) && xs.unify(this.tail);
}
Nil.prototype.unifyCons = function(x, xs) {
  return false;
}
```

Then, we can write programs which use *unification* — very powerful tool available in logic programming languages such as Prolog. For example,

```
var ws = unknown();
var xs = known(cons(known("foo"), ws));
var y  = unknown();
var ys = known(cons(y,
            known(cons(known("bar"),
                  known(cons(known("baz"), known(nil)))))));
```

After "xs.unify(ys)" is evaluated, "y.toString()" evaluates to "foo" and "listToString(ws)" is "[bar,baz]".

It is known that operators to handle first-class continuations such as call/cc in Scheme [9], especially those for composable continuations (shift and reset) [3] can make backtracking expressible [6] — backtracking is another powerful tool in logic programming languages. If both first-class continuations and dynamic prototype are available, we will be able to simulate Prolog programs in ECMAScript and to apply the standard list library functions to the results of such a code fragment.

5 Conclusion and Future Work

ECMAScript can be already regarded as a (strict) functional language. — it provides λ-expressions (anonymous functions) and the ability to add new operations later to existing objects. With a small extension, we have shown that, it can also simulate more easily other programming paradigms such as lazy functional and logic programming paradigms. So far, to the author's knowledge, studies on multi-paradigm programming have often concentrated on simulation of control structures such as lazy evaluation and back-tracking. However, simulation of data structures is equally important. The extension proposed in this paper facilitates simulation of data structures peculiar to other programming paradigms.

The idea is just to allow the prototype property to be computed dynamically from other properties of the object. It is even possible to create a new object dynamically and to use it as a prototype as is done for Delay.

A similar idea is proposed by the author [8] as an extension of the system of polymorphic variants (extensible algebraic datatypes) for a purely functional language Haskell. The idea is to convert a value into another variant during pattern-matching when a matching pattern does not exist. Though lazy data structures do not need to be simulated in Haskell since it is already a lazy language, it helps to simulate an imperative program in a purely functional language. It is interesting that similar ideas can be used in both purely functional and object-oriented languages to simulate other programming paradigms.

There are some points to be worked on, however. First, efficiency must be considered. Most ECMAScript implementations seem to use a *cache* in order to make property look-up fast. If prototype is computed dynamically, use of a cache is not so much effective. However, it is likely that as for data structures which use dynamic prototypes, the prototype chains tend to be short. Therefore, hopefully, lack of cache has a very little effect on efficiency.

It would be possible to adopt part of the proposal of this paper to class-based object-oriented languages at value level by simply specifying how the object is transformed when the current class does not have a necessary method. However, most class-based languages such as Java are strongly-typed and then we also need type checking. To type wrapper constructors such as `Delay`, at least, we would need the notion of parametric (generic) classes. Therefore, the design of a type checking algorithm is not trivial.

Though we have seen that ECMAScript can be used as a functional language, the most distinct difference with genuine functional languages such as ML [11] and Haskell [13] is that of the type systems. In ML and Haskell, type inference is done at compile time and no type error will take place at run-time. Designing a type inference algorithm or a soft typing algorithm [17] for (a subset of) ECMAScript including dynamic prototyping would be an interesting future research topic.

ECMAScript is often misunderstood as a dirty and ad-hoc language. However, it is the interface to the real world (*e.g.* the browser) that is dirty and we can say its core is a tidy object-oriented language. If it is strengthened properly (probably with facilities to help debugging), it may be even suitable for an educational purpose since it supports multi-paradigm programming to a certain extent.

References

[1] ECMAScript language specification, 3rd edition, 1999. available form `http://www.ecma.ch/`.

[2] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) 1990*, 1990.

[3] Andrzej Filinski. Representing monads. In *Annual ACM Symp. on Principles of Prog. Languages*, January 1994.

[4] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, 1998.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] Ralf Hinze. Prological features in a functional setting axioms and implementations. In *Third Fuji International Symposium on Functional and Logic Programming*, pages 98–122, 1998.

[7] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.

[8] Koji Kagawa. Shrinkable polymorphic variants. In *Proceedings of JSSST Workshop on Programming and Programming Languages (PPL 2002)*, March 2002. (in Japanese).

[9] Richard Kelsey, William Clinger, Jonathan Rees, et al. Revised^5 report on the algorithmic language Scheme, Feb 1998.

[10] Xavier Leroy, Damien Dolegez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system, release 3.00 Documentation and user's manual.* INRIA, April 2000.

[11] Robin Miler, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML – Revised.* MIT Press, 1997.

[12] Lawrence C. Paulson. *ML for the Working Programmer – 2nd Ed.* Cambridge University Press, 1996.

[13] Simon Peyton Jones, John Hughes, et al. *Haskell 98: A Non-strict, Purely Functional Language*, February 1999. `http://www.haskell.org/onlinereport/`.

[14] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. organzing programs without classes. *List and Symbolic Computation*, 4(3), 1991.

[15] David Ungar and Randall B. Smith. SELF: the power of simplicity. *Lisp and Symbolic Computation*, 4(3), 1991.

[16] Philip Wadler, Walid Taha, and David MacQueen. How to add laziness to a strict language, without even being odd. In *Workshop on Standard ML*, Baltimore, September 1998.

[17] Andrew K. Wright and Robert Cartwright. A practical soft type system for scheme. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 250–262, June 1994.

[18] Matthias Zenger and Martin Odersky. Extensible algebraic datatypes with defaults. In *Proceedings of the International Conference on Functional Programming*, September 2001.

A case in Multiparadigm Programming : User Interfaces by means of Declarative Meta Programming

S. Goderis [⋆], W. De Meuter, J. Brichau

Programming Technology Lab, Vrije Universiteit Brussel, Belgium

Abstract. Because there is currently no formal way to specify user interfaces, nor a clean way to decouple a user interface from its application code, we propose in this position paper the use of Declarative Meta Programming (DMP) to solve these problems. DMP uses facts and rules to write down a user interface in a declarative way, and will provide a more formal way to specify user interfaces. Furthermore DMP cleanly separates user interface from application code. The Declarative Meta Programming language SOUL that we intend to use, combines the declarative paradigm (for the user interface specification) and the object-oriented paradigm (for the application code). This position paper thus describes a case in multiparadigm programming.

1 Introduction

Although our knowledge concerning software engineering tasks has grown considerably during the last 20 years, few of these techniques are applied onto User Interfaces. Currently we distinguish two major problems with User Interfaces, namely

- the lack of a clean way to separate and couple the user interface and its underlying application, and
- the lack of a clear, powerful and uniform formalism to specify user interfaces.

Solving the first problem will benefit the independent evolution of the user interface and the application code. Even 'clean' separations like the Model-View-Controller pattern still have the problem that the underlying model code is interspersed with 'changed-messages'. A lot of user interface builders provide graphical tools for the development of user interfaces, but mainly they only generate stubs for procedures and methods. These stubs have to be coded manually by the software engineer in order for the user interface and the application to be coupled. Changing the user interface then results in manually changing this generated, and often unreadable, code. Current user interface builders lack a clean way to couple the user interface with the application code.

[⋆] Author is financed with a doctoral grant from the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT)

Solving the second problem will allow a non-programmer to build complete user-interfaces containing all necessary functionality that should otherwise be programmed. Furthermore, a uniform formalism for specifying user interfaces will benefit the porting of user interfaces to various graphical platforms (pc, mac, web, handhelds, mobiles,...). Since the desired formalism is to specifically express the *description* of user interfaces, a declarative formalism would be the most appropriate. A declarative *programming* language would even be better because it provides an executable declarative formalism and, as such, provides a user-interface specification language. The major advantage of using a complete language for specifying user interfaces is that it permits to write down more powerful descriptions of the dependencies and relations between the different user interface components. It also permits us to create abstractions from the low-level descriptions of user interface components to high-level descriptions of compositions of user interface components. Having different levels of abstractions will allow to easily replace one of the levels with a new set of declarations. Lower levels will be more platform dependent, and changing platforms will result in changing these lower levels while the higher levels can be kept. Thus there will be no need to rewrite the user interface specification.

In this position paper we state that creating a user interface for an application is a multiparadigm problem where

- the user interface is to be written down declaratively,
- the model is to be coded manually, and
- the coupling (how and where) of the user interface with the model is to be written down declaratively.

This approach requires the object-oriented paradigm for coding the model, and the declarative paradigm for writing down the user interface and the coupling. We will thus use a multiparadigm programming language combining both paradigms, namely a Declarative Meta Programming language. At Programming Technology lab several researchers have been using this paradigm for several purposes, such as code generation [2], co-evolution between design and implementation [8], aspect-oriented programming [1, 5], component-based development [7], etc. One of the artifacts build for this paradigm is the Smalltalk Open Unification Language (SOUL) which is the one we will focus on for our approach.

In section 2 we introduce the SOUL declarative programming language, which we will use in section 3 to specify and generate user interfaces. We conclude in section 4.

2 Smalltalk Open Unification Language

SOUL is a declarative meta layer on top of Smalltalk. It provides a prolog-alike programming language [4] and thus has all the properties of a declarative language. Logic facts are used to write down data or knowledge, while rules are used to reason about these facts and derive new facts. Furthermore SOUL is implemented as a meta layer on top of Smalltalk, and it provides reflection and introspection operators [9]. Therefore it is possible to access the underlying Smalltalk system for retrieving information and adapting the underlying system according to the descriptions and rules at the upper level. This implies that, based on the rules and facts of the SOUL level, it is possible to generate code on the Smalltalk level. For more details (and the syntax) we refer to [8].

The Quoted term is a logic term in SOUL that we will most extensively use for our approach. It is a special logic term that was specifically included in SOUL to ease the manipulation of program source code by logic programs. It is similar to the quasi-quoted list in Scheme, which also represents programs as datastructures. The quoted term allows for writing down Smalltalk code as it is without it being evaluated. This construct can contain any kind of strings, possibly with some logic variables that are 'filled in' by the logic inference process. Templates (quoted terms with source code and logic variables) in combination with the substituted variables will result in 'real' code. Slightly changing the facts and rules will generate different code. In the following example firing the query `addClass` will result in a quoted term (between curly braces) containing the Smalltalk code that, if executed, will add a subclass `Matrix` to the super class `Array`.

```
addNewClass(Array, Matrix).

addClass({?super subclass: #?className }) if
   addNewClass(?className, ?super),
   class(?super)
```

3 DMP for User Interfaces

Model-based user interface development environments (MB-UIDEs) provide a context where developers can design and implement user interfaces in a systematic way, and more easily than when using traditional user interface development tools [6]. To achieve this aim, MB-UIDEs allow to describe the user interfaces through the use of declarative models. Pinheiro names three major advantages when using declarative user interface models [6] :

- A more abstract description of the user interface is provided ;
- User interfaces can be modelled using different levels of abstraction; the models can be refined incrementally; and user interface specifications can be re-used ;
- tasks related to the user interface design and implementation processes can be automated.

The idea behind MB-UIDE's is to be able to specify, generate and execute user interfaces.

As mentioned before we also want to use a declarative approach, namely Declarative Meta Programming, for specifying and generating user interfaces.

Since we tend to use SOUL, we intend to have a application coded in Smalltalk. SOUL itself will be used to write down the user interface specification and the coupling between user interface and application in a declarative way. We distinguish three parts of logic code, namely

– facts representing the user interface specification,
– facts representing the hooks in the application code, and
– rules expressing how to generate the code to combine both parts.

Hooks in the application code are certain points where calls to the user interface can be made, or where event handling (calls from the user interface) can be taken care of. Together with the coupling rules these hooks will ensure that the user interface is plugged onto the application. The user interface specification itself can consist of different layers, which allows for different abstraction levels.

3.1 User Interface specification

We will have to determine how the specification of different kinds of user interfaces can be put into a logic format, i.e. by the use of facts and rules. User interface specifications can range from elementary and simple knowledge (e.g. a title) to very advanced specifications (e.g. this circle always has to be centered in that rectangle). Elementary specifications are for example HTML and XML specifications, often used for web based user interfaces. Transforming these syntax-trees (what they basically are) into logic facts can easily be done by integrating a simple parser into SOUL. More advanced specifications that are represented by user interface builders can be transformed unambiguously into logic facts by means of transforming the structures used by these builders. Another kind of advanced specifications are constraints, which are, up to a certain degree, easily expressible by the use of logic programming.

For example a simple declaration for a user interface component `button` can be :

```
button(?name, ?label).
style(?name, ?style).
size(?name, ?size).
color(?name, ?color).
```

This specification indicates that there is a button with a name `?name` which has a label, a style (font of the label), size and a color.

A Formalism for User Interface specifications Declarative user interface specifications provide a more formal way of writing down user interfaces. This kind of formal specification is a major disadvantage of user interface builders these days. Graphical user interface builders provide a limited set of components which can not be altered by the user. Typically these builders use their own internal data structures, and the provided ‘drawingtools’ on top of these structures are not a sufficient formal medium to specify user interfaces. A declarative approach combines the powerfulness of programming languages with the intuitivity of graphical tools. We do acknowledge that as an end tool a graphical user interface is wished for, but it can easily generate logic declarations.

For example, because SOUL is a programming language we can specify a rule for the previous example that declares all button properties to be obligatory :

```
isButton(?aButton) if
   button(?aButton, ?),
   style(?aButton, ?),
   size(?aButton, ?),
   color(?aButton, ?).
```

Different abstraction levels DMP also allows for different levels of abstraction. For the specification of user interfaces this is important because it will lead towards a more clean specification, but it also allows for easier generation of different kinds of user interface code. For instance we can specify a simple user interface consisting of a title, a textbox and two buttons :

```
title(uiTitle, ‘This is a simple user interface’).
textbox(goOn, ‘Do you want to continue?’).
button(yesButton, ‘yes’).
button(noButton, ‘no’).

color(uiTitle, green).
color(yesButton, grey).
color(noButton, grey).
size(yesButton, 2cm).
```

`color` and `size` specify properties of these lower level (possibly visual) components.

For often used combinations of components we can specify higher level concepts such as questions, actions, dependencies, listings, etc. For instance, the textbox and two buttons from the previous example express the higher level concept *question.* In logic this question can be specified in terms of a textbox and buttons :

```
question(q, ‘do you want to continue?’).

textbox(?name, ?questionText) if
   question(?q, ?questionText).
button(?name, yes) if
   question(?q, ?questionText).
button(?name, no) if
   question(?q, ?questionText).
```

The `textbox` and `button` rules are the transformation between the two layers of abstraction.

DMP also allows more powerful declarations. For instance the use of variables allows to express similar specifications by one specification only. For example, changing the color of the whole user interface to blue can easily be expressed by

```
color(? , blue).
```

instead of rewriting this fact for each element of the interface. This is extremely handy if we want to create a consistent user interface with the same 'look and feel' for multiple subcomponents.

We can also decide to let a property of a certain component depend on the property of another component. In

```
size(aTextbox, ?size) if
   size(aWindow, ?size).
```

the size of `aTextbox` depends on the size of `aWindow`. By the use of quoted terms it is possible in SOUL to let the size be a piece of Smalltalk code that will have to be executed at the time the size is really needed. For `aWindow` we could retrieve the size from the underlying Smalltalk system with

```
size(?aWindow, {?model  windowSize}) if
   model(?aWindow, ?model).
```

A more powerful example illustrating the same idea is the following :

```
relativeSize(?comp1, ?comp2, 0.5).

size(?comp1, {?model windowSize * ?ratio}) if
   relativeSize(?comp1, ?comp2, ?ratio),
   model(?comp2,?model).

size(?comp2, {?model windowSize}) if
   relativeSize(?, ?comp2, ?),
   model(?comp2,?model).
```

It expresses the dependency between two components where `comp1` has 0.5 times the size of `comp2`. Calculating the actual size of these components then depends on this relative size, and a size we supposedly retrieve from a class bound to `?model`.

Facts and rules referring to the underlying system are part of the coupling between the user interface specification and the underlying application code. These kind of rules and facts would have to reside at the lowest level of the user interface specification because it implies that it is known we are using Smalltalk user interface code. Using DMP for the generation of another kind of user interface code implies changing these levels. The lower level abstractions will be more user interface dependent (platform-dependent).

3.2 Generating User Interface code

As was stated by De Volder, DMP naturally supports Generative Programming [3]. After all we do want to be able to generate user interface code based on the user interface specification, whether it be HTML, Smalltalk code or something else.

In order to generate the user interface code, we will need to annotate the application code with logic facts which will represent the hooks onto which to plug the user interface. Using the code generation techniques of DMP, both user interface and application code will be linked. Since user interface configuration resides at the logic level, launching a query will generate the right code based on this configuration. Slightly changing the configuration will result in the generation of different code. The kind of user interface code that has to be generated will depend on the interface. For web based user interfaces it is possible that generating HTML or XML is sufficient, other interfaces will require a more advanced model (e.g. a Model-View-Controller pattern). These interfaces are independent from the underlying application and the different parts can be evolved and maintained rather independently from one another.

As an example, consider following rules that are part of a logic program that generates Smalltalk user interfaces from our specification in logic facts and rules:

```
method(?model, openView,
          { win := ?kindOfWindow new.
            win label: ?title.
            ?subwindowsAdditionCode win open}) if
   baseWindow(?uiName,?kindOfWindow),
   title(?uiName,?title)
   model(?uiName,?model),
   findall(?subwindowCode, subWindow(?uiName,?subwindowCode),
            ?subwindowsAdditionCode)


subWindow(?superId,
          {win addWindow: (?kindOfWindow new)
            atPositions: #(?x1,?x2,?y1,?y2). }) if
   embed(?superId,?subId,?x1,?x2,?y1,?y2)
   kindOfWindow(?subId,?kindOfWindow).
```

Launching the query `if method(?class,?methodName,?code)` will return all necessary methods that need to be generated in the Smalltalk application to build the user interface that we specified. A kind of code-generator will do exactly this and compile the resulting code. How the user interface window, with possible subwindows is to be created, is specified by using the quoted term. The kind of window that is to be generated, its title and model are specified by other logic facts. The same holds for the subwindow's position and kind of window.

We would like to stress that this is a tentative example and our approach needs to be elaborated. However, the same kind of technique was applied on components by De Volder [2].

4 Conclusion

Currently there is no way to easily specify uster interfaces formally, nor is there a clean way to decouple a user interface from its underlying application. In this position paper we propose the use of Declarative Meta Programming as a solution for these problems. DMP allows to write down user interface specifications in a declarative way by the use of facts and rules, and thus gives us a more formal way to write down user interfaces. In addition DMP cleanly separates the user interface from its underlying application and provides a means to generate the user interface code coupled with this application. Building user interfaces in this way is a case of multiparadigm programming. The declarative paradigm is combined with the object-oriented paradigm, and user interface specification is decoupled from the user interface code, and decoupled from the application code. We used SOUL as a multiparadigm programming language to put this into effect.

References

[1] J. Brichau. Declarative meta programming for a language extensibility mechanism. In *ECOOP 2000, Workshop on Reflection and Meta Level Architectures*, 2000.

[2] K. De Volder. *Type-Oriented Logic Meta Programming.* Phd thesis, Programming Technology Lab, Vrije Universiteit Brussel, September 1998.

[3] K. De Volder. Generative logic meta programming. In *ECOOP 2001, Workshop on Generative Programming*, 2001.

[4] P. Flach. *Simply Logical.* John Wiley and sons, 1994.

[5] K. Gybels. *Aspect-Oriented Programming using a Logic Meta Programming Language to express cross-cutting through a dynamic joinpoint structure.* Bachelors thesis, Programming Technology Lab, Vrije Universiteit Brussel, August 2001.

[6] P. Pinheiro da Silva. User Interface Declarative Models and Development Environments: A Survey. In P. Palanque and F. Paternò, editors, *Proceedings of DSV-IS2000*, volume 1946 of *LNCS*, pages 207–226, Limerick, Ireland, June 2000. Springer-Verlag.

[7] M. J. Presso. *Reflective and Metalevel Architecture in Java: from Object to Components.* Master thesis, Programming Technology Lab, Vrije Universiteit Brussel, 1999.

[8] R. Wuyts. *A logic meta-programming approach to support the co-evolution of Object-Oriented design and implementation.* Phd thesis, Programming Technology Lab, Vrije Universiteit Brussel, January 2001.

[9] R. Wuyts and S. Ducasse. Symbiotic reflection between an object-oriented and a logic programming language. In *ECOOP 2001 International workshop on Multi-Paradigm Programming with Object-Oriented Languages*, 2001.