

1st European Lisp Symposium (ELS'08)

Bordeaux, France, May 22-23, 2008

Preface

In the last couple of years, we have seen a growing interest in the Lisp programming language and its various dialects, including Common Lisp, Scheme, ISLISP, Dylan, and so on. Several user group meetings, workshops and conferences have been organized in recent years, with great success. Especially in Europe, but also elsewhere, Lisp is gaining momentum.

With the European Lisp Symposium, we aim to start a series of annual events that is especially suitable for novel research results, but also for insights and lessons learned from practical applications and education perspectives, all involving Lisp dialects. For this year's symposium, we have received 15 submissions, and after a careful review process, the program committee selected seven of them for presentation at the main track of the symposium. Furthermore, we have received seven additional submissions for a work-in-progress track, which describe ongoing work that is not ready for publication yet. Those work-in-progress papers will be discussed at the symposium using a writers' workshop format for giving feedback to the authors in a dedicated session. This volume contains the papers accepted for both the main track and the work-in-progress session. Some of the papers of the main track are selected for further review after the symposium and will be considered for publication in a future special issue of the Journal of Universal Computer Science (J.UCS).

The success of the 1st European Lisp Symposium was only possible because of the great efforts of many people. I would especially like to thank the local organizer Robert Strandh for providing the facilities of the University of Bordeaux 1 and the local organization team for taking care of the many important details that are necessary to run such an event.

Pascal Costanza, Brussels, May 2008

Program Committee

- Marco Antoniotti, Universita Milano Bicocca, Italy
- Marie Beurton-Aimar, Université Bordeaux 1, France
- Jerry Boetje, College of Charleston, USA
- Theo D'Hondt, Vrije Universiteit Brussel, Belgium
- Irène Durand, Université Bordeaux 1, France
- Marc Feeley, Université de Montréal, Canada
- Erick Gallesio, Université de Nice / Sophia Antipolis, France
- Rainer Joswig, Independent Consultant, Germany
- António Leitão, Technical University of Lisbon, Portugal
- Henry Lieberman, MIT, USA
- Scott McKay, ITA Software, Inc., USA
- Ralf Möller, Hamburg University of Technology, Germany
- Nicolas Neuss, Universität Karlsruhe, Germany
- Kent Pitman, PTC, USA
- Christophe Rhodes, Goldsmiths College, University of London, United Kingdom
- Jeffrey Mark Siskind, Purdue University, USA
- Didier Verna, EPITA Research and Development Laboratory, France

Organizing Committee

- Antoine Allombert
- Marie Beurton-Aimar
- Irène Durand
- Nicole Lun
- Robert Strandh

Sponsors

Franz, Inc.
555 12th Street, Suite 1450
Oakland, CA 94607
USA
<http://www.franz.com>



LispWorks Ltd.
St. John's Innovation Centre
Cowley Road
Cambridge
CB4 0WS
England
<http://www.lispworks.com>



Table of Contents

Main Track

Irène A. Durand, Sylviane R. Schwer Reasoning about qualitative temporal information with S-words and S-languages	1
Sebastián González, Kim Mens, Alfredo Cádiz Context-Oriented Programming with the Ambient Object System	17
Mikael Laurson, Mika Kuuskankare Visual Programming in PWGL	33
António Menezes Leitão UCL-GLORP - An ORM for Common Lisp	47
Timothy Moore An Implementation of CLIM Presentation Types	63
Jim Newton, Christophe Rhodes Custom Specializers in Object-Oriented Lisp	75
Didier Verna Binary Methods Programming: the Clos Perspective	91

Work-in-Progress Track

Carlos Agon, Jean Bresson, Gérard Assayag OpenMusic: Design and Implementation Aspects of a Visual Programming Language	107
Marco Antoniotti CLAZY: Lazy Calling for Common Lisp	125
Jerry Boetje, David Williams, Robert Shields, Hector Raphael Mojica, Seth Rylan Gainey CLforJava 2008 - Work-in-Progress Report	133
Jónathan Heras, Vico Pascual, Julio Rubio, Francis Sergeraert Improving the usability of Kenzo, a Common Lisp system for Algebraic Topology	155
Jim Newton Vns - Name Space Facility	177
Mikhail Semenov Prime-Lisp 2.0: an ISLisp Implementation in .NET with Multithreading Extensions	185
Pierre Thierry, Simon E.B. Thierry Abolishing object-oriented xenophobia: designing highly reusable libraries	199

Main Track

Reasoning about qualitative temporal information with S-words and S-languages

Irène A. Durand

LaBRI, Université de Bordeaux,
idurand@labri.fr

Sylviane R. Schwer¹

LIPN, Université Paris-Nord
schwer@lipn.univ-paris13.fr

Abstract: Reasoning about incomplete qualitative temporal information is an essential topic in many Artificial Intelligence applications. In the domain of natural language processing for instance, the temporal analysis of a text yields a set of temporal relations between events in a given linguistic theory. Our aim is first to situate the events with respect to each other and to describe (compute or count) all possible relations between them. We first present the formalism of S-languages which formally describes this domain. We explain why Lisp is adequate to implement this theory. Next we describe a Common Lisp system SLS (for S-LanguageS) which implements part of this formalism. A graphical interface written using McCLIM, the free implementation of the CLIM specification frees the potential user of any Lisp knowledge. A complete example illustrates both the theory and the implementation.

1 Introduction

The notion of time is ubiquitous in any activity that requires intelligence. In particular, several important notions like change, causality, and action are described in terms of time. Time has been recognized as a fundamental notion in modeling and reasoning about changing domains. Reasoning about temporal constraints is thus an important task in many areas of computer science and elsewhere, including in scheduling, natural language processing, planning, database theory, diagnosis, circuit design, archeology, genetics, and behavioral psychology [DGV05].

Many frameworks for formalizing time have been proposed, all based on work by logician philosophers who were concerned with physics or language theories, among whom Frege, Prior, Montague, Hamblin, Reichenbach, or Russell, Whitehead and Nicod. This explains why all works have been handled in a logical framework.

In this article, we are concerned with the qualitative aspect of temporal reasoning, *i.e.* only how "objects" are time-related to each other, without information about any quantitative aspect. We are thus interested in two problems:

(i) a representation problem: how to represent time or temporal objects and what temporal relations are to be represented and how,

¹ Supported by the project "ANR Blanc Conique".

(ii) a calculus problem for the reasoning: knowing that a and b are in relation r_1 and b and c are in relation r_2 , what possible relations are derived for a and c .

According to the classical spatial representation of time, on a geometrical oriented line, temporal items are taken as points, intervals or chains or points/intervals, depending on whether objects to be represented are viewed as event-like, lasting or iterative. For each representation type, an algebra has been proposed: the point algebra [vBC90] for expressing the three basic relations between points on a line, the point-interval algebra [Vil82] for expressing the five basic relations between a point and an interval on a line, the interval algebra [Ham69] for the thirteen basic relations between intervals on a line, which has become well known since the appearance of [All83]. Other suggested calculi have been derived from one or several of the ones cited above. These algebra are integrated with various logics. There are three known ways of representing and reasoning about temporal information: first order logic, modal logics, and temporal relational calculi. All these approaches are restricted to binary relations and based on transitivity tables like the one for point-point algebra shown in the next table which is read in the following way: the first column shows one of the basic relation between two points p_1 and p_2 on an oriented line: precedes ($<$), equals ($=$) and succeeds ($>$), the first line shows the same for two points p_2 and p_3 , and an inside cell provides the possible derived relations between p_1 and p_3 . For instance, if $p_1 < p_2$ and $p_2 > p_3$, we can't derive any constraint between p_1 and p_3 . But if $p_1 < p_2$ and $p_2 < p_3$, then necessary, we have $p_1 < p_3$.

	$p_2 < p_3$	$p_2 = p_3$	$p_2 > p_3$
$p_1 < p_2$	$p_1 < p_3$	$p_1 < p_3$	$p_1 < p_3$ $p_1 = p_3$ $p_1 > p_3$
$p_1 = p_2$	$p_1 < p_3$	$p_1 = p_3$	$p_1 > p_3$
$p_1 > p_2$	$p_1 < p_3$ $p_1 = p_3$ $p_1 > p_3$	$p_1 > p_3$	$p_1 > p_3$

A qualitative temporal constraint in this framework is depicted in terms of a graph, whose vertices are labeled with temporal objects, and arcs with temporal relations. The consistency of such a graph depends on the calculus with transitivity tables and there is no way to directly express a n-ary relation between n objects.

The use of graphs entailed the resolution of path-consistency and particular complexity problems which gave rise to the exhibition of some subsets of relations (convex, pointizable, Ord-Horn, *i.a.* [NB95]).

The S-languages formalism is based on a totally different approach. It was first introduced in [Sch02]. Its aim was to propose lighter and more intuitive representation than the one given in [Lig91] itself an extension of [Vil82].

Following the natural philosophy of Whitehead, Nicod and Russell, which traces

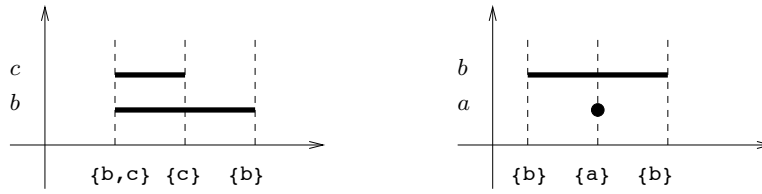


Figure 1: Representation of S-words w_1 and w_2

back to Leibniz², a letter a is associated with each temporal object (event, or fact or state) a and acts as its identity. Any object related to a determined scale of time or a point of view can be described as event-like, lasting or repetitive. Let us denote the way the object is to be perceived as its temporal *aspect*. The aim of S-language is to provide a uniform framework for describing both the temporal aspect of objects and their temporal relationships.

If a is depicted as event-like, only one occurrence of its identity will be used; if a is depicted as lasting, two occurrences of its identity will be needed, each of them representing one bound of its interval of duration. If it is depicted as lasting and iterating n times, it will be represented by $2n$ occurrences of its identity.

In a word (a sequence of letters), the fact that a letter b is after another letter a expresses *precedence* between a and b . To express simultaneity, we define *S-letters* (S for *Set* languages in general or for *Synchronization* in the framework of time), which are sets of letters occurring at the same time. *S-words* are words over S-letters. Each occurrence of the identity of an object will appear at most once in an S-letter, which is assumed to model a moment.

The S-word $w_1 = [\{b, c\} \{c\} \{b\}]$ contains several pieces of information: there are two temporal objects b and c ; they are both lasting objects; b and c start at the same time and b finishes after c . The S-word $w_2 = [\{b\} \{a\} \{b\}]$ means that the temporal object a is event-like and occurs during the lasting object b . A temporal representation of w_1 and w_2 is given in Figure 1.

Given a set of temporal objects, S-words can express constraints over them. Our work focuses on the problem of expressing, enumerating or counting possible scenarii given a set of temporal constraints.

To simplify the presentation of this work, we shall restrict ourselves to handle **lasting** objects – that is lasting and repetitive lasting objects – although taking into account event-like objects does not induce major difficulties.

² These philosophers asserted that time is built from nature and that a moment is a "passage of the nature" [Whi20], that is, the set of all events occurring simultaneously at that moment.

2 Preliminaries

2.1 Letters and S-letters

Each temporal object e is represented by a *letter* e . By α , we denote the *alphabet* of all letters. It is supposed to be linearly ordered according to the order of the letters in their enumeration. For instance, $\alpha = \{a, b, c\}$ and $a < b < c$. $\#\alpha$ denotes the cardinality of α .

An *S-letter* is a non-empty subset of α . It defines synchronization points between events. For instance, $\{a, c\}$ is an S-letter meaning that a and c occur simultaneously. By $S_\alpha = \mathcal{P}(\alpha) \setminus \{\emptyset\}$, we denote the set of S-letters whose *underlying* alphabet is α and name it the S-alphabet of all letters. For instance,

$$S_{\{a,b,c\}} = \{\{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\}\}.$$

2.2 S-words

An *S-word* is a sequence of the S-letters, in other words an element of $(S_\alpha)^*$. We surround the sequences of the S-letters of an S-word by brackets $[\]$. The *Parikh vector* of a S-word w is the vector \vec{w} of $\mathbb{N}^{\#\alpha}$ whose i^{th} coordinate is the number of occurrences of the i^{th} letter. The alphabet $\alpha(w)$ of a S-word w is the set of letters appearing in its S-letters.

Example 1. let $\alpha = \{a, b, c, d\}$ and $w = [\{a\} \{a,b\} \{a,c\} \{a,b,c\}]$.
 $\alpha(w) = \{a, b, c\}$ and $\vec{w} = (4, 2, 2, 0)$.

Letters in S-letters of an S-word w can be *marked* with the following bijective marking. For all letters $l \in \alpha$ appearing in w , the first occurrence of l is marked 0, the second 1 and so on. Marking the S-word w gives:

$$[\{a_0\} \{a_1, b_0\} \{a_2, c_0\} \{a_3, b_1, c_1\}].$$

A possible meaning for S-words is the following. The marked letter l_0 (and each other appearance of l with an even mark) indicates that the object associated with the letter l starts. Each l_i with an odd mark indicates that the object stops. This is illustrated in Figure ??.

As the marking of an S-word is bijective, we generally don't write marks. However, when dealing with subwords of S-words — which happens when handling incomplete temporal descriptions — it will be informative to write the marks. For instance, given two lasting objects a and b , "b starts strictly after the end of a" could be written by the complete S-word $[\{a\} \{a\} \{b\} \{b\}]$

(which is implicitly $[\{a_0\} \{a_1\} \{b_0\} \{b_1\}]$)

or only described by $[\{a_1\} \{b_0\}]$ as $[\{a_0\} \{a_1\}]$ and $[\{b_0\} \{b_1\}]$ are implicit.

Hence our letters in S-letters are always marked (either implicitly or explicitly). The *marked-alphabet* $\alpha_M(L)$ of an S-word w is the set of marked (or implicitly marked) letters of w .

2.3 S-languages

An *S-language* is a set of S-words. Given an alphabet α , and a vector $\vec{w} \in \mathbb{N}^{\#\alpha}$ (which associates an integer to each letter), the set of all possible S-words is called the *S-universe* of α and \vec{w} and is denoted by $\mathcal{U}(\alpha, \vec{w})$. Given an S-language L , the alphabet $\alpha(L)$ of L is the set of letters of L and the marked-alphabet $\alpha_M(L)$ of L is the set of marked-letters appearing in the S-words of L .

If, by hypothesis, we know that each object associated with a letter l occurs a finite number of times n then for each letter l , we have a maximum index of $2n - 1$. In that case, S-words have *finite* length and the S-universe of interest is the one associated with the vector having as its i^{th} coordinate the length of the S-word that depicts it. In these cases, we do not mention the Parikh vector of the S-universe.

But S-languages are not always restricted to a finite S-universe: in [Sch07a] S-languages over infinite S-words are used to deal with execution traces in distributed systems. In this case the alphabet is finite but the alphabet of marked-letters is not (the set of possible marks is infinite) and the Parikh vector cannot be defined, but the S-universe is defined as the set of all possible S-words and is infinite.

An S-language will be represented either in *extension*, *i.e.* by giving the list of its S-words (this is possible in the finite case only: finite language of finite S-words) or by expressions over S-languages, which are called *S-expressions* (not to be confused with `Lisp` Sexpressions) using operators. Operations and expressions over S-languages will be presented in Section 3.

Suppose for instance that we have two independent objects a and b , each occurring once. They are represented by S-words $[\{a\}\{a\}]$ and $[\{b\}\{b\}]$ respectively. The S-universe is the S-language containing all the possibilities of combining these two objects that is all the S-words having $(2, 2)$ as Parikh vector. This S-language has 13 S-words, given by the Delannoy number $D(2, 2)$ [Slo], which depicts the 13 possible relationships between two intervals on a line and well-known in artificial reasoning community as Allen's relations [All81, All83]. We shall see in Section 3 that this S-language can be represented by the *mix* of the two S-words: $[\{a\}\{a\}] X [\{b\}\{b\}]$.

So, for just *two* objects a and b , each occurring once and without any specific constraint (other than "the beginning of an object occurs strictly before its end"), we have a set of 13 possibilities (S-words) for combining them. Now if constraints exist between the objects, we will get an S-language which is a subset of these 13 possibilities. Each subset of the S-universe corresponds to specific constraints.

Example 2. For instance, if we add the constraints that b must start strictly after a and end after than or at the same time as a , we get the following S-language with 4 possibilities: $L_1 = \{[\{a\}\{b\}\{a,b\}], [\{a\}\{a\}\{b\}\{b\}], [\{a\}\{a,b\}\{b\}], [\{a\}\{b\}\{a\}\{b\}]\}$.

There are 2^{13} S-languages included in the S-universe; the whole part represents the absence of constraint; the empty part represents incompatible constraints.



Figure 2: The 13 relations between two intervals on a line

3 Operations and expressions on S-languages

3.1 Classical operations on languages

From one point of view, S-languages are a special case of formal languages. Consequently, all classical operations on formal languages apply [RS96]. In particular, the boolean operations (union, intersection, complement), concatenation, mirror are defined in the usual way considering that S-letters are the letters of the S-words. In the classical framework, letters are basic objects which cannot be decomposed. In the S-languages framework, the letters of the S-words are S-letters, *i.e.* sets of letters which we may want to compose or decompose. Expressions over S-languages will be referred to as S-expressions (not to be confused with Lisp S-expressions (Sexpr). The classical projection would be to project over a sub-alphabet of S-letters: it erases S-letters.

3.2 S-projection

In the S-language framework, we may define the S-projection over a sub-alphabet of letters which erases letters inside the S-letters of a S-word. The same extension can be considered for morphisms and inverse morphisms.

The *S-projection* of an S-word w over the alphabet α , denoted by $w|_{\alpha}$ is the S-word obtained by erasing from w all occurrences of letters which are not in α and then every S-letter which has become empty.

Example 3. Let $w = [\{a, c\} \{a, b\} \{c, d\} \{a, b, c\}]$ and $\alpha = \{a, b\}$.
 $w|_{\alpha} = [\{a\} \{a, b\} \{a, b\}]$.

The S-projection of an S-language is the set of the S-projections of its S-words.

3.3 The join operation

Consider two S-languages L_1, L_2 over respective alphabets $\alpha(L_1)$ and $\alpha(L_2)$. Each S-language L_i represents temporal constraints which restrict the S-universe $\mathcal{U}(\alpha(L_i))$. *joining* the two languages L_1 and L_2 consists in constraining $\mathcal{U}(\alpha(L_1) \cup \alpha(L_2))$ with the union of the constraints of both languages. The join operation will be denoted by the symbol \mathcal{J} .

Example 4. Recall $L_1 = \{[\{a\}\{b\}\{a,b\}], [\{a\}\{a\}\{b\}\{b\}], [\{a\}\{a,b\}\{b\}], [\{a\}\{b\}\{a\}\{b\}]\}$ of Example 2. The language $L_2 = \{[\{a\}\{a,c\}\{c\}]\}$ can be described by the constraint "c starts when a stops". $L_1\mathcal{J}L_2$ yields the S-language $\{[\{a\}\{b\}\{a,c\}\{b,c\}], [\{a\}\{b\}\{a,c\}\{b\}\{c\}], [\{a\}\{b\}\{a,c\}\{c\}\{b\}], [\{a\}\{b\}\{a,b,c\}\{c\}], [\{a\}\{b\}\{b\}\{a,c\}\{c\}]\}$.

There are two special cases for the join operation: the first case occurs when the alphabets of the two languages are identical, then the join corresponds to the intersection of the two languages; the second case occurs when the alphabets are disjoint and is described below.

3.3.1 The mix operation (join with disjoint alphabets)

In the case of disjoint alphabets, the join operation is a kind of *shuffle* that we call *mix* and denote by \mathbf{X} : it considers all possibilities of ordering independent letters.

In the case where $L_1 = [\{a\}\{a\}]$ and $L_2 = [\{b\}\{b\}]$, the S-language corresponding to $L_1\mathcal{J}L_2$ (already seen in Section 2.3) can be obtained by applying the two rewrite rules

$$\begin{aligned} \{a\}\{b\} &\rightarrow \{a,b\} \\ \{a,b\} &\rightarrow \{b\}\{a\} \end{aligned}$$

on the concatenation of L_1 and L_2 which is $[\{a\}\{a\}\{b\}\{b\}]$. The lattice (shown in Figure 3.3.1) obtained by applying the rewrite rules contains all the S-words of $L_1 \mathbf{X} L_2$. Note that these S-words are the same as the one in Figure 2.3.

This principle generalizes to any number of letters and S-languages with any cardinality.

A mix expression is a compact way of representing an S-universe (all the possibilities for a given set of temporal objects). S-universes are usually very big (so big that we can't compute them in practice) so the mix is an indispensable tool to handle S-languages. Very often the computation of the language corresponding to a mix expression will lead to a combinatorial explosion. Consequently, such computation should be avoided as much and as long as possible. The idea is to first perform every possible simplifications which could prune part of the search space.

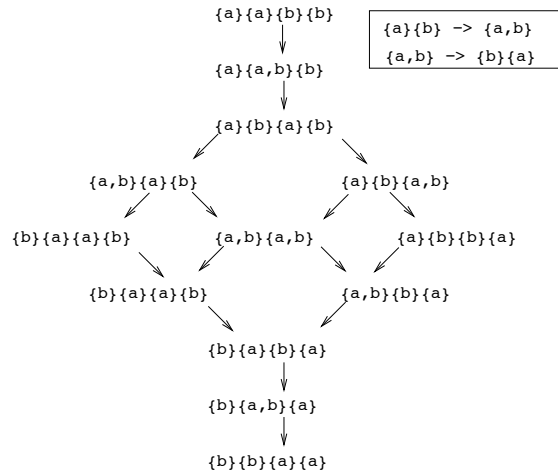


Figure 3: Lattice of the mix operation

3.3.2 Join operation (with intersecting alphabets)

In the general case, the alphabets have a non-empty intersection ($\alpha(L_1) \cap \alpha(L_2) \neq \emptyset$).

The basic operation is defined on S-words. Let f and g two S-words.

If $\alpha(f) \cap \alpha(g) = \emptyset$ then $fJg = fXg$ as defined above. Otherwise, let $\beta = \alpha(f) \cap \alpha(g) \neq \emptyset$. If the projections $f|_\beta$ and $g|_\beta$ differ then the constraints inherent to the two words are incompatible and $fJg = \{\}$. Otherwise, the S-words are compatible and fJg is the S-language containing all S-words h written over $\alpha(f) \cup \alpha(g)$ which satisfy $h|_{\alpha(f)} = f$ and $h|_{\alpha(g)} = g$. Let for instance

$f = [\{a, c\}\{a, b\}\{c, d\}\{a, b, c\}]$ and

$g = [\{e\}\{a, e, f\}\{e\}\{a, b\}\{f\}\{a, b, f\}\{e\}]$.

Then $\beta = \{a, b\}$, $f|_\beta = [\{a\}\{a, b\}\{a, b\}] = g|_\beta$ and

$fJg = \{[\{e\}\{a, c, e, f\}\{e\}\{a, b\}\{c, d, f\}\{a, c, b, f\}\{e\}],$
 $[\{e\}\{a, c, e, f\}\{e\}\{a, b\}\{c, d\}\{f\}\{a, c, b, f\}\{e\}],$
 $[\{e\}\{a, c, e, f\}\{e\}\{a, b\}\{f\}\{c, d\}\{a, c, b, f\}\{e\}]\}$.

However, we can give a more compact representation using the mix operation:

$[\{e\}\{a, c, e, f\}\{e\}\{a, b\}] \cdot ([\{c, d\}] \times [\{f\}]) \cdot [\{a, b, c, f\}\{e\}]$

The join operation extends to languages: the join of two S-languages is the union of the joins of an S-word of the first language and an S-word of the second. A description of the algorithm can be found in [Sch07b]. Our implementation provides both a recursive and an iterative version of it. The join algorithm is a crucial in the S-languages setting because solving a problem described by a set of constraints $\{E_1, E_2, \dots, E_n\}$ consists in computing the S-language corresponding to the S-expression

$E = E_1 J E_2 J \dots J E_n$.

3.4 Example

The following example is inspired by [Rev96]. Consider a set of 6 trains named $\{A, B, C, D, E, F\}$ with the following set of temporal constraints.

1. A, B and E reach the platform at the same time
2. A leaves before B.
3. A leaves after or at the same time as C but before the arrival of D.
4. D and F arrive at the same time as B is leaving.
5. E and D leave at the same time.

We consider the following problem: how many platforms are necessary to satisfy constraints 1 to 5. We formalize the problem into the S-languages framework. For each train, we consider the event corresponding to the time during which the train remains at the platform. Because of security reasons, we do not allow that a train to arrive on a track from which a train is currently leaving.

Our alphabet is $\alpha = \{a, b, c, d, e, f\}$, one letter for each train. The S-universe is the S-language represented by the following mix expression

$[\{a\}\{a\}] \times [\{b\}\{b\}] \times [\{c\}\{c\}] \times [\{d\}\{d\}] \times [\{e\}\{e\}] \times [\{f\}\{f\}]$
 which means that we have 6 lasting temporal objects. The S-universe contains

$$D(2, 2, 2, 2, 2, 2) = D(2^6) = 308682013$$

S-words [Slo]. The five constraints can be expressed by the following five S-expressions:

1. $E1 = [\{a, b, e\}] \cdot ([\{a\}] \times [\{b\}] \times [\{e\}])$
2. $E2 = ([\{a\}] \times [\{b\}]) \cdot [\{a\}\{b\}]$
3. $E3 = (([\{a\}] \times [\{c\}\{c\}]) \cdot [\{a\}\{d\}\{d\}]) \cup$
 $(([\{a\}] \times [\{c\}]) \cdot [\{a, c\}\{d\}\{d\}])$
4. $E4 = [\{b\}\{b, d, f\}] \cdot ([\{f\}] \times [\{d\}])$
5. $E5 = ([\{e\}] \times [\{d\}]) \cdot [\{d, e\}]$

3.5 Simplifying S-expressions

For solving a set of constraints $\{E1, E2, \dots, En\}$, one must evaluate the S-expression $E1 \cup E2 \dots \cup En$. In general, it is not tractable to evaluate the S-languages L_i corresponding to the E_i and then joining them because the intermediate S-languages are much too big. The key idea is to simplify to e until it becomes reasonable to compute the final S-language. Finding simplifications and proving they are correct is a difficult domain which is not completely explored. The first kind of simplifications results from classical properties of the operators like associativity, commutativity, idempotence and distributivity. The other simplifications concern the join operation or its special cases (mix, intersection). For instance, the intersection of two languages with disjoint alphabets is empty; the join of a language with its S-universe is the language itself.

For our trains example of Section 3.4, SLS is able to simplify the S-expression which evaluates to an S-language containing 24 S-words of length between 5 and 7. The final language can be written usig mix as:

$$E = ([\{c\}\{c\}] \times [\{a,b,e\}]) \cdot [\{a\}] \cdot [\{b,d,f\}] \cdot ([\{f\}] \times [\{e,d\}]) \times [\{a,b,e\}] \cdot [\{a,c\}] \cdot [\{b,d,f\}] \cdot ([\{f\}] \times [\{e,d\}])$$

In order to solve our problem, we have to recall the good interpretation of what this S-language depicts (the possible relationships between the periods where trains are stopped at a platform), then to find inside E an S-word which minimizes the meeting or interleaving between these periods. The first choice is to take $([\{c\}\{c\}\{a,b,e\}])$ from the left sub-S-expression $([\{c\}\{c\}] \times [\{a,b,e\}])$ which isolates the train C. The new S-expression is $E' = [\{c\}\{c\}\{a,b,e\}\{a\}\{b,d,f\}] \cdot ([\{f\}] \times [\{e,d\}])$ and contains only 3 S-words. First, C stops and leaves, then A, B, E arrive all at the same time, then we need at least three tracks. But A leaves only before the arrival of D and F, then we need one more track. The answer of the problem is then that 4 tracks are enough; 4 tracks are also sufficient for all S-words of E' .

4 Implementation of S-languages

It will not take long to justify the choice of the Common Lisp language to implement the theory of S-languages: the domain is typically symbolic as opposed to numeric; the data are highly hierarchical which justifies an object-oriented language; in addition, multiple inheritance is very useful for factoring properties and associated methods for simplifying S-expressions.

4.1 Implementation of basic objects

The basic SLS objects are letters (`letter`), marked letters (`mletter`), S-letters (`sletter`), alphabets for all the different kinds of letters (`alphabet`, `malphabet`), S-words (`sword`).

To prevent combinatorial explosion we use the well-known technique of *hash-consing*: each element of each object category is represented by a unique Lisp object; there is a list for each category of object; the objects are stored in the list corresponding to its category. When the creation of an object is required, a look-up is done in the corresponding list; if an object with equal components (in the `eq` sense) is found such object is returned; otherwise a new object is constructed and stored in the list. Here is the example of the `mletter` case.

```
(defmethod make-mletter ((string string) &optional (mark 0))
  (let* ((letter (make-letter string))
        (name (name letter)))
    (or (find-object name (mletters *spec*)
                    :test (lambda (name mletter)
                          (and (eq name (name mletter))
                               (= mark (mark mletter))))))
        (let ((mletter (make-instance 'mletter :letter letter
                                     :mark mark)))
          (setf (mletters *spec*)
                (append (mletters *spec*) (list mletter)))
                mletter))))
```


This technique has also the advantage that SLS basic objects are `eq-comparable` which improves time performance.

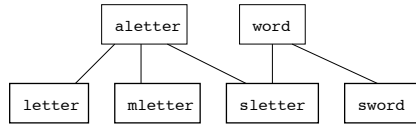


Figure 4: Classes for basic SLS objects

The hierarchy of the classes describing basic SLS objects is presented in Figure 4.1. Note that an S-letter, being a sequence of letters, is itself a word (but not an S-word).

4.2 Implementation of S-expressions

The class `alanguage` contains all objects which describe languages. A language can be represented by its set of words (`language`, `word`) or by an expression. An expression is defined recursively: it is either a concrete `language` or an expression with an operator and whose arguments are expressions. Note the use of the `mixin`

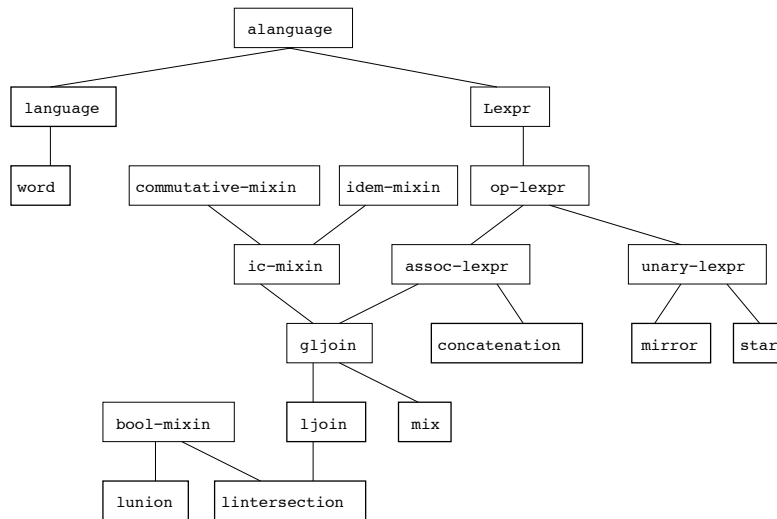


Figure 5: Class hierarchy for representing S-languages

classes to capture properties which help simplifying expressions. For instance, the pri-

mary method `clean-args` normalizes the arguments of an associative S-expression. The secondary methods complete this task according to the other properties of an operator. For instance, if the operator is idempotent, we can remove duplicated or equivalent arguments.

```
(defmethod clean-args ((lexpr assoc-lexpr) ...)
  (defmethod clean-args :before ((lexpr fold-mixin))
    (setf (args lexpr)
          (remove-duplicates (args lexpr) :test #'equivalent))
    lexpr)
```

4.3 Specifications for SLS

SLS handles a set of specifications that can be loaded interactively. A specification consists of a signature, possibly a set of variables, followed by a list of SLS objects. SLS objects are S-words, S-expressions, S-languages, Problems (set of S-expressions which correspond to constraints). In a same specification, one stores objects from a common S-universe.

Figure 4.3 shows an example of such a specification. That specification contains the train problem of Section 3.4. It also shows how to specify S-word, S-expressions or S-languages in extension.

```
Problem trains
  ([{a,b,e}] . ([{a}] X [{b}] X [{e}]))
  (([{a}] X [{b}]) . [{a}{b}])
  ((([{a}] X [{c}{c}]) . [{a}{d}{d}]) U
   ([{a}] X [{c}]) . [{a,c}{d}{d}]))
  ([{b}{b,d,f}] . ([{f}] X [{d}]))
  ([{e}] X [{d}]) . [{d,e}]]

Sword [{c}{c}{a,b,e}{a}{b,d,f}{e,d}{f}]
Sexpr ([{a}{a}] X [{b}{b}])
Sexpr ([{a,b}] J [{b 1,c}])
Slanguage L [{[{a}{b}{a,b}], [{a}{a}{b}{b}], [{a}{a,b}{b}],
              [{a}{b}{a}{b}]]
```

Figure 6: Example of an SLS specification

4.4 The graphical interface

A graphical user interface helps the user load his/her data (S-words, S-expressions, S-languages) and apply operations on it. It is written using the McCLIM[SM02] system which is the free implementation of the CLIM specification. A snapshot of the SLS window after loading the `train.txt` specification is shown Figure 7. All the commands are either accessible from the command line in the top window or from

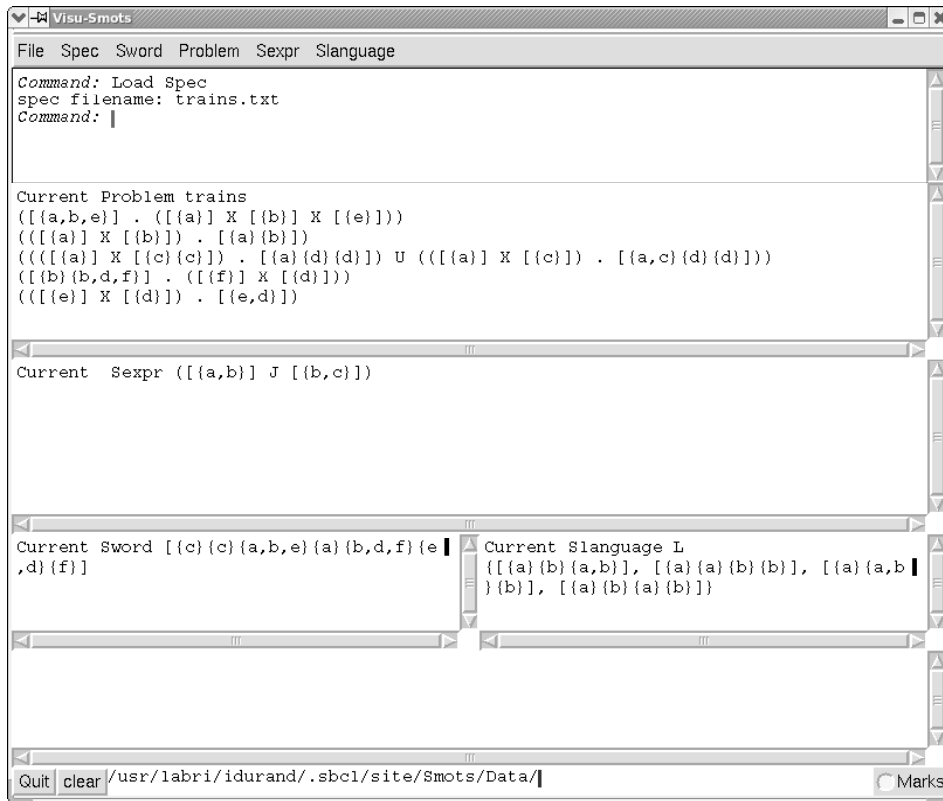


Figure 7: First snapshot of SLS

menus, classified according to the type of object they operate on. Here we have applied the command `Solve` (also in the `Problem` menu) which transforms the set of constraints of the problem into a (when possible) simplified S-expression which becomes the current S-expression. Next we have applied the `Slanguage Sexpr` command (also in the `Sexpr` menu) which computes the S-language corresponding to the current S-expression and invoked the `Cardinality Slanguage` command (also in the `Slanguage` menu) which prints the cardinality of the current S-language. Finally, with the `Membership To Slanguage`, we verify that the current S-word belongs to the current S-language. The final look of the window is shown in Figure 8.

SLS contains altogether 6000 lines of `Common Lisp` of which around 1200 correspond to the graphical interface. On the project page, <http://dept-info.labri.u-bordeaux.fr/~idurand/SLS/>, one can find a description of the project, a User's Manual, an archive with the latest source and executable files for a few architectures.

do for S-word, S-letters. Furthermore, we also plain to analyse in terms of S-expressions, the convex, pointizable and Ord-Horn classes studied in the interval algebra theory [NB95].

At the outside level, much work needs to be done to allow non-computer scientists to use the tool. Representing graphically S-words could be a first step. Next we could think of a tool for helping the user defining graphically constraints between objects resulting in a set of S-words.

Acknowledgements

The authors would like to thank the referees for their constructive reports and Lucas Saiu for his careful rereading.

References

- [All81] James F. Allen. An interval-based representation of temporal knowledge. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 221–226, 1981.
- [All83] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.
- [AS03] Jean-Michel Autebert and Sylviane R. Schwer. On generalized delannoy paths. *Journal on Discrete Mathematics*, 16(2):208–223, 2003.
- [DGV05] Mickael D. David, Dov M. Gabbay, and Lluis Vila. (eds). Elsevier, 2005.
- [Ham69] C. L. Hamblin. Starting and stopping. *The Monist*, 53(3):410–425, 1969.
- [Lig91] Gérard Ligozat. On generalized interval calculi. In *AAAI*, pages 234–240, 1991.
- [NB95] Bernhard Nebel and Hans-Jürgen Bürckert. Reasoning about temporal relations: A maximal tractable subclass of allen’s interval algebra. *Journal of the ACM*, 42(1):43–66, 1995.
- [Rev96] Joel Revault. *Une modélisation par le graphe de la relation meet pour traiter des contraintes temporelles exprimées à l’aide d’intervalles*. Phd thesis, Université de Nantes, 1996.
- [RS96] G. Rozenberg and A. Salomaa. *Handbook of Formal Languages: Word, Language, Grammar*, volume 58 of *Lecture Notes in Computer Science*. Springer, 1996.
- [Sch02] Sylviane R. Schwer. S-arrangements avec répétition. *Comptes Rendus de l’Académie des Sciences, Mathématiques*, 4:261–266, 2002.
- [Sch07a] S. Schwer. Temporal reasoning without transitive tables. arXiv:0706.1290v1 [cs.AI], June 2007.
- [Sch07b] Sylviane R. Schwer. Traitement de la temporalité des discours : une analysis situs. In *Information temporelle, procédures et ordre discursif*, volume 18 of *Cahiers Chronos*. Rodopi, Amsterdam, 2007.
- [Slo] Neil Sloane, editor. *The On-Line Encyclopedia of Integer Sequences*, chapter A055203. <http://www.research.att.com/njas/sequences/>.
- [SM02] Robert Strandh and Tim Moore. A free implementation of clim. In *Proceedings of the International Lisp Conference*, San Francisco, California, October 2002.
- [vBC90] P. van Beek and R. Cohen. Exact and approximate reasoning about temporal relations. *Computational Intelligence*, 6:132–382, 1990.
- [Vil82] Marc Vilain. A system for reasoning about time. In *Proceedings of the AAAI*, pages 197–201, 1982.
- [Whi20] Allan North Whitehead. *The concept of nature*. Cambridge University Press, Cambridge, 1920.

Context-Oriented Programming with the Ambient Object System

Sebastián González, Kim Mens, Alfredo Cádiz

Département d'ingénierie informatique

Université catholique de Louvain

sebastian.gonzalez|kim.mens|alfredo.cadiz@uclouvain.be

Abstract In this paper we present *AmOS*, the Ambient Object System that underlies the *Ambience* programming language. *AmOS* implements a computation model that supports highly dynamic behaviour adaptation to changing contexts. It is developed entirely in Common Lisp. Apart from being purely object-based, *AmOS* features fully reified closures and multimethods, and a subjective dispatch mechanism for method lookup. We claim that these features make it a very simple and elegant paradigm for context-oriented programming.

Key Words: context-oriented programming, subjective dispatch, multiple dispatch, prototype-based programming, ambient intelligence

Category: D.3.3 [Programming Languages]: Language Constructs and Features

1 Introduction

The introduction of mobile devices equipped with sensors and wireless network provisions allow for present-day mobile applications to become aware of their environment and to interact with it. As a simple example, modern laptops adjust their backlit keyboard and screen brightness dynamically, thanks to an ambient light sensor. At the software level, service discovery protocols such as DNS-SD¹ have set the stage for service-oriented architectures in mobile networks, such that printers and file servers can be found on the fly, for example. Using these sensors and mobile network infrastructure, far more advanced application interactions and adaptations than the ones just mentioned can be envisaged [7].

However, the kind of advanced dynamic behaviour adaptation that would fully exploit the potential of mobile systems requires adequate programming language support. To allow applications to change their behaviour in different contexts, context-specific behaviour should not be hard-wired in the application logic under the form of conditional statements scattered across method bodies, nor by using dedicated design patterns like *Visitor*, *State* and *Strategy* [8].

The need for adequate programming abstractions that enable application context-awareness has given rise to Context-Oriented Programming [9, 11, 12]. Our approach follows the same direction. We propose a computation model that supports highly dynamic behaviour adaptation without hard-coded, crosscutting

¹ DNS Service Discovery, see <http://www.dns-sd.org/>

conditional code, and that avoids the use of dedicated software architectures or design patterns.

Whereas our model has been presented in the past using a Smalltalk-like surface syntax [10], its core has been written, and is therefore readily available, in Common Lisp.² We call this core the Ambient Object System (*AmOS*). In essence, *AmOS* is a prototype-based object layer built on top of Common Lisp, featuring multimethods and subjective dispatch [15]. *AmOS* does not rely CLOS, in particular because *AmOS* does not have a notion of *class* [14].

In complement to a previous paper [10] where we illustrated the main features of our Ambience language and how they support run-time adaptation of mobile applications to changing contexts, in this paper we concentrate on the inner workings of the underlying object system *AmOS* and discuss its advantages for context-oriented programming.

To give the reader a first feel of the language before diving into the core abstractions of our model, the following section introduces a simple *AmOS* program that will serve as running example throughout the paper.

2 Motivating example

The example illustrates how the behaviour of a mobile phone can be programmed and made adaptable to the context. We deliberately do not explain the detailed semantics of the language constructs used in this example, but rely on the reader's intuition instead. In the forthcoming sections we revisit this example as we gradually introduce the different language features in more detail.

The example concentrates on functionality related to receiving and advertising calls on mobile phones, with the following requirements. Urgent calls are treated with priority over normal calls. Incoming calls can be advertised by playing a ringtone or by activating a built-in vibrator. The choice between the two depends on the current environment: the ringtone is used by default, whereas the vibrator should be used in silent places like museums, libraries and situations such as meetings. Calls received while the user is sitting inside a car should mute the car's radio and be advertised on the car's speakers.

One of the key features of *AmOS* is the support of first-class contexts. Contexts are objects representing physical or logical properties of the environment in which the system is running. These properties may be about the user, the machine, the surroundings or in general any information which is computationally accessible [11], be it acquired through sensor input, network communication, generated internally, or otherwise.

In our example, we first create a `@telephony` context, representing a prototypical situation in which a telephony service is available. Inside a mobile phone

² See <http://ambience.info.ucl.ac.be>

such service always is:

```
(defcontext @telephony)
```

By convention, prototype names are prefixed with the `@` symbol. The `@telephony` context thus created is a plain object, without any special status in comparison to other objects in the system.

Next we proceed to define objects and behaviour that are specific to the telephony context. For the sake of the example, a phone object simply contains a list of incoming calls and a speaker on which to advertise those calls:

```
1 (with-contexts (@telephony)
2   (defproto @phone (clone @object))
3   (add-slot @phone 'incoming-calls (list))
4   (add-slot @phone 'speaker 'phone-speaker)
5   (defproto @mobile-phone (extend @phone)))
```

For simplicity, we use a symbol to identify the speaker, but in a fully developed application, the speaker would be a more complex object with suitable behaviour. In line 5 the result of `extend` is an empty object that delegates to the object being extended.³ As a result, all behaviour that is not understood directly by `@mobile-phone` will be handed over to `@phone`.

Still in the telephony context, we define a phone call as an object that can be received on any phone:

```
(with-contexts (@telephony)
  (defproto @phone-call (clone @object))
  (defmethod receive ((call @phone-call) (phone @phone))
    (advertise call phone)
    (add-incoming call phone))
  (defmethod advertise ((call @phone-call) (phone @phone))
    (format t "Playing ringtone through ~a" (speaker phone)))
  (defmethod add-incoming ((call @phone-call) (phone @phone))
    (enqueue call (incoming-calls phone))))
```

The `receive` multimethod is specialised on both `@phone-call` and `@phone`. It encodes the prototypical behaviour for receiving calls on a phone: the call is advertised and added to the list of incoming calls. The `advertise` method encodes the prototypical way of announcing a call to the user, i.e. by playing a ringtone. The `add-incoming` method encodes the prototypical way of treating an incoming call, i.e. by enqueueing it to the phone's list of incoming calls.

AmOS methods, even when belonging to the same context, can be overloaded by using the same name but different specialisers. For example, behaviour that is better suited for urgent calls can be defined by overloading `add-incoming` as follows:

³ For a discussion of delegation in prototype-based languages and how it differs from class-based inheritance, see the seminal paper by Lieberman [13] and the book edited by Noble et al. [14].

```
(with-contexts (@telephony)
  (defproto @urgent-call (extend @phone-call))
  (defmethod add-incoming ((call @urgent-call) (phone @phone))
    (push call (incoming-calls phone))))
```

This version of `add-incoming`, specially conceived for urgent calls, puts the call in the front of the incoming call queue instead of at the end. Overloaded multimethods permit defining behaviour that is better suited to certain kinds of objects.

In addition to having explicit dependencies on their argument kinds, *AmOS* methods have an implicit dependency on the context in which they are defined, and thus can be overloaded on that context as well. This will be explained next.

Functionality that is specific to a car context can be defined as follows:

```
(defcontext @car)
(with-contexts (@car)
  (defproto @radio (clone @object))
  (defmethod mute ((device @radio))
    (format t "Muting radio~%")))
```

In the context of cars with a radio on board, the default behaviour of the `advertise` method can be specialised so that the car's speaker is used instead of the phone's built-in speaker, after the car's radio has been muted:

```
1 (with-contexts (@telephony @car)
2   (add-slot @phone 'speaker 'car-speaker)
3   (defmethod advertise ((call @phone-call) (phone @phone))
4     (mute @radio)
5     (resend)))
```

This version of the `advertise` method is specific to the *combination* of the `@telephony` and `@car` contexts. Whereas the `@telephony` context is inherent to the phone and is always active, the `@car` context is activated or deactivated dynamically when the user enters or leaves a car. The behaviour just defined will be exhibited only when the phone is in car context. The `resend` message in line 5 is like `call-next-method` in CLOS: it invokes the next most-specific version of the currently executing method. Note in line 2 that a context-specific slot is added to `@phone`, for which a `speaker` accessor method will be defined in the `(@telephony @car)` context combination. When this combination is inactive, the original `speaker` accessor method (and thus original slot value) will be used.

All code shown so far is written at development time and deployed into the phone. During normal use, actual mobile phones and phone calls are created by cloning the respective prototypes, and behaviour is triggered by invoking multimethods like `receive`:

```
(let ((bobs-phone (clone @mobile-phone))
      (alices-call (clone @urgent-call)))
  (receive alices-call bobs-phone))
```

the default output will be:

Playing ringtone through PHONE-SPEAKER

whereas in car context the output of the same expression will be:

Muting radio
Playing ringtone through CAR-SPEAKER

Note that the call to `receive` is not surrounded by any context-switching construct such as `with-context`. In *AmOS*, hard-coding context switch points in the source code is discouraged except for the definition of prototypical objects and their behaviour. Context switches aimed at adapting system behaviour at run time are supposed to be performed orthogonally to the base code.⁴ This point is discussed further in Section 6.

3 *AmOS* Core Concepts

AmOS aims at being a multiparadigm model that does not sacrifice simplicity and homogeneity for expressiveness and flexibility. Section 2 gave a first glimpse of that from an end-user perspective. In the remainder of the paper we show that simplicity and homogeneity are at the core semantics of the object model. We start by highlighting the underlying concepts that have been introduced in an intuitive fashion so far. These concepts form the cornerstones of the object model, on which all the rest is based.

Objects Every first-class entity in *AmOS* is an object — that is, the model is *purely* object-based. The observable properties of objects are their identity, acquaintances and behaviour. Whereas identity is an immutable (defining) characteristic, acquaintances and behaviour can vary over time. The latter two thus constitute the state of an object.

Some objects in the system act as representative examples of domain entities, and are therefore called *prototypes*. However, prototypes do not have a special status in the language other than being meaningful exemplars [13, 14].

Cloning New objects can be created by cloning existing ones. Cloned objects have a distinct, unique identity, but their acquaintances and behaviour are copied (shallowly) from the cloned object.

Messages Interaction among objects happens through message passing. A message is a request for interaction among the participants involved in the message. To this effect, each message has a *selector* object that identifies the desired interaction, and an argument list of objects that will take part in it. Messages are *symmetric*: there is no distinguished receiver for any given message.

⁴ Basically, a separate context management thread is in charge of performing actions such as (`activate-context @car`) when such change is detected in the outside world.

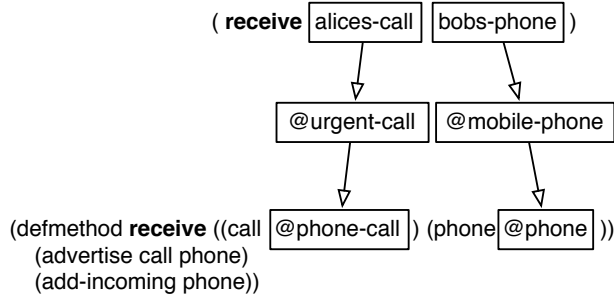


Figure 1: Method applicability for a given message. The hollow-headed arrows denote delegation relationships.

Delegation Behaviour can be delegated from one object to another by placing a delegation link between them. When we refer to *inheritance* in this paper we mean such delegation-based inheritance. Since objects can have multiple delegations, a directed graph of delegation links can be formed. Messages that are not understood by an object can be handled by one of the delegates in the delegation graph. Cyclic delegations are supported, as explained in Section 5. Sample delegations are shown in Figure 1.

Methods Methods describe prototypical interactions among objects. Every method has a selector that identifies the particular interaction it implements, and a list of prototypical objects that take part in the interaction. The method is said to be *specialised* on those particular objects.

Rather than belonging to a single class as in Java or to a single generic function as in CLOS, *AmOS* methods belong simultaneously to all their specialisers. In other words, method ownership is shared, both at a conceptual and technical level. Methods are thus *symmetric*, just like messages are.

Because of shared ownership, a method can be accessed only if the client holds references to suitable arguments and suitable contexts to which the method is applicable. In contrast, generic functions in CLOS are globally visible objects giving access to all homonym methods.

Method applicability For any given message, a method is *applicable* if the selector and arguments of the message match those of the method. The selectors match if they have the same object identity. The arguments match if each message argument delegates in zero or more steps to the method specialiser in the same position, as illustrated in Figure 1.

Method specificity Due to multiple inheritance, more than one method might

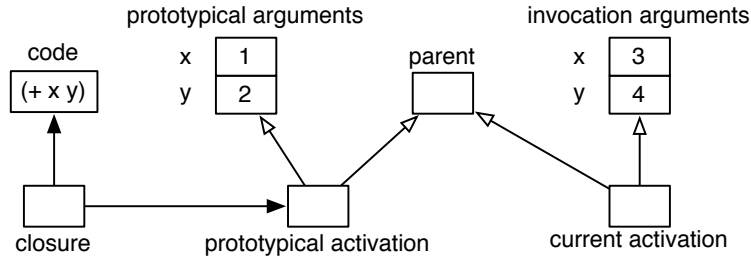


Figure 2: Prototypical activation and cloned activation with actual arguments. Solid arrows represent object references, the hollow arrows represent delegations.

be applicable for any given message. A notion of *specificity* is introduced to solve ambiguities, which is a strict, total order relationship among methods. A second source of ambiguity is multiple dispatch. To solve this kind of ambiguity, *asymmetric dispatch* [4] is used, giving earlier message arguments more importance during dispatch than later arguments. With these rules there will always be a method that is more specific than the others and can therefore be chosen for execution.

These concepts are all there is to the basic computation model of *AmOS*. Perhaps the least trivial part is message disambiguation. This topic is explained further in Section 5. The next sections progressively show how the core concepts just explained are sufficient to support the fundamental constructs of our model, which in the end enable dynamic behaviour adaptation to context.

4 Closures and Activations

The most basic executable entity in *AmOS* is the *closure*. It has `lambda`-like syntax and semantics, as the following example illustrates:

```
(& (x y) (+ x y)) → closure
```

Every closure has an associated *activation record*—hereafter simply called *activation*— which holds the dynamic information that is associated with its invocation. Activations are the environments in which closure code is executed.⁵ Like in Self [3], activations are first-class objects.

It is possible to specify prototypical argument values to be held in the activation of a closure. They are placed next to each argument name:

```
(& ((x 1) (y 2)) (+ x y)) → closure
```

This closure is illustrated in Figure 2. As can be seen, the prototypical activation

⁵ In stack-based execution models, activations are also known as *stack frames*.

delegates to an *arguments* object, which holds one slot per closure argument. Upon invocation, the closure activation is cloned and the prototypical arguments are substituted by the invocation arguments. The closure’s code is then executed in this freshly created environment and is thus fully reentrant. Figure 2 shows the fresh activation resulting from the following invocation:

```
(invoke (& ((x 1) (y 2)) (+ x y)) (list 3 4)) → 7
```

Each activation delegates to a *parent* object, also illustrated in Figure 2. Messages not understood by the current activation or by its arguments object are delegated to the parent. The parent corresponds to the enclosing lexical scope of the closure, so that outer definitions can be seen inside the closure’s environment. For the particular case of the *top-level activation*, which has no enclosing lexical environment, the parent is the so-called *current context*. This context link is crucial to our approach and is explained further in Section 6.

As shown in this section, the semantics of closures involves nothing more than objects, cloning and delegation. The next section explains methods and their dispatch infrastructure.

5 Methods and Specialisation

Methods are obtained by enriching closures with a dispatch mechanism. Since methods are extended forms of closures, the execution semantics described in Section 4 applies unmodified to methods. In the case of methods, the prototypical arguments are considered to be *argument specialisers*. The code of the method is designed to work for those specialisers in particular, and for any extension (through delegation) thereof. Reconsider for instance the `receive` method:

```
(defmethod receive ((call @phone-call) (phone @phone))
  (advertise call phone)
  (add-incoming call phone))
```

The `receive` method is basically a named closure with prototypical arguments `@phone-call` and `@phone`, which are used as specialisers. The link between a method and its specialisers is established through *roles*, originally proposed in the Prototypes with Multiple Dispatch model [15]. Any object that is used as method specialiser plays a role in the interaction described by the method. As illustrated in Figure 3, the argument specialiser objects `@phone-call` and `@phone` play a role in the `receive` interaction, at the first and second positions respectively. The illustrated roles are triplets (s, i, m) of the selector s identifying the interaction, the position i at which the object plays the role, and the method m implementing the behaviour.

Figure 3 also shows the conceptual difference among the different kinds of objects. Objects in the *plain* layer correspond to concrete domain entities that are being manipulated at the moment; objects in the *prototypes* layer are prototypes

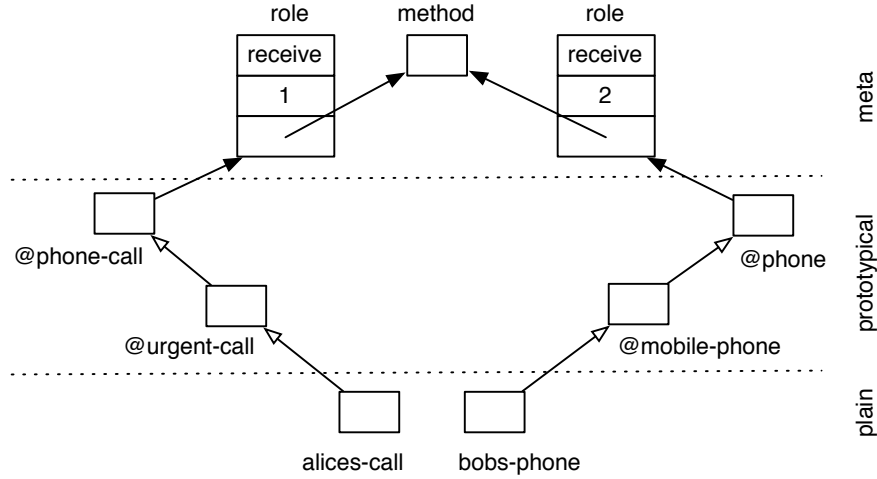


Figure 3: Roles corresponding to the `receive` method specialised on the `@phone-call` and `@phone` prototypes, and arguments `alices-call` and `bobs-phone` for which the method is applicable.

(usually meant for cloning, rather than direct manipulation); finally, the core computation model is available through a series of *meta* objects describing base objects, their roles, methods, and so on.

Method overloading brings about the problem of choosing the method version that is best suited to the given arguments. Specificity among applicable methods is defined by *rank vectors* [15]. Each rank vector entry contains the delegation distance between the message argument and corresponding method specialiser. For instance, the rank vector of the method illustrated in Figure 3 for the message with arguments `alices-call` and `bobs-phone` is (2, 2), since the path in the delegation graph that goes from message argument to method specialiser is of length 2 for both arguments. As another example, the version of the `add-incoming` method specialised on urgent calls (see Section 2) has rank vector (1, 2), since `alices-call` is one hop away (delegation-wise) from `@urgent-call`, and `bobs-phone` is two hops away from `@phone`. A rank vector with only zeroes is a “perfect match”, corresponding to the case where the message arguments are precisely the method specialisers.

We use an adapted version of the C3 linearisation algorithm [1] to topologically sort the delegation graph of each message argument and have a well-defined notion of distance. Our adaptation of C3 supports delegation cycles trivially, by taking into account only the first occurrence of a delegate in the linearisation and ignoring any further occurrences arising from cycles. Despite our handling

of cycles, we still have to devise an automatic resolution strategy for inconsistent delegation graphs (that cannot be linearised by C3 [1]). Such automatic strategy is necessary in *AmOS*, as ambiguities cannot always be detected at development time due to dynamic inheritance. Delegation graphs can change arbitrarily at run time, and chances for ambiguous cases are higher than in systems with static inheritance.

Ambiguities arising from multiple dispatch — for example, considering whether the rank vector $(1, 2)$ is more specific than $(2, 1)$ — are precluded by imposing left to right argument precedence as in CLOS (i.e. a lexicographic ordering): $(1, 2)$ is thus considered more specific than $(2, 1)$. As a consequence, methods with a better match in earlier argument positions will be considered more specific than other applicable methods. This choice is justified by observing that important arguments tend to have earlier argument positions, while more auxiliary arguments are usually placed rightwards; the extreme case is observed in languages with single dispatch, in which only the leftmost argument is dispatched dynamically and therefore completely determines selected behaviour.

Method specialisation is useful in defining behaviour for special kinds of objects and dealing with particular cases without hard-coding conditional statements. The next section explains the way we further exploit specialisation and multiple dispatch to define *context-specific* behaviour, and the way such behaviour can be adapted dynamically as needed.

6 Context-Oriented Programming in *AmOS*

Run-time behaviour adaptation is supported in *AmOS* by introducing a kind of dynamic scoping mechanism for methods. Generally speaking, the main reason why dynamic scoping is useful is that it allows the caller’s state to influence the behaviour exhibited by the callee in a deep fashion (i.e. across nested method calls). Such influence is not intertwined in the form of arguments that must be passed from one function or method to the next. Clearly, having such kind of arguments is quite inconvenient, as the arguments crosscut all methods and messages that need to be influenced [5], and all possible influences that might prove useful need to be foreseen and hard-coded. Dynamic scoping can help alleviating these problems.

AmOS identifies dynamic scoping — a concept coming mainly from the functional programming world — with subjective behaviour — a concept coming from the object-oriented world [16], which unfortunately has faded into oblivion until now. Subjective behaviour is roughly equivalent to dynamic scoping: it is behaviour that depends on the caller’s point of view or state. As Smith and Ungar observe [16], any language with multiple dispatch can easily support subjective behaviour by passing with every message an implicit argument that represents

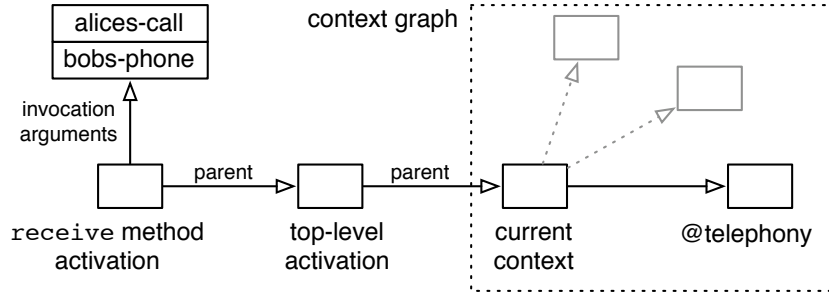


Figure 4: Invocation of the `receive` method.

the current point of view or state of the caller. This implicit argument participates in the dispatch process as any other argument does. As a result, chosen behaviour will depend on this implicit subjective element.

In *AmOS*, the *current activation* of the executing closure or method⁶ is passed implicitly as first argument of every message. This way, behaviour selection will depend on the current execution environment of the sender. This simple exploitation of multiple dispatch results in a kind of dynamic scoping mechanism that is surprisingly convenient, as we will illustrate in the remainder of this section.

For any given message, applicable methods are first looked up in the current activation, and by following the lexical parent link, they are looked up further in enclosing lexical scopes, until the top-level activation is reached. Rather than stopping at this point by having an empty object be the parent of the top-level activation, we assign a plain object which we consider the *current context*. The current context can delegate further to other context objects as needed. Figure 4 shows a sample configuration of activations and context objects corresponding to the invocation of the `receive` method. Activation parent links correspond to enclosing lexical scopes and are therefore kept constant, in correspondence to the program text structure. Delegation links starting from the current context object and beyond are dynamically managed and may change at run time. Hence, messages that are not understood by the static activation chain will be delegated to the current context. The objects that are reachable by delegation starting from the current context constitute the *current context graph* or simply *context graph* (shown in the dashed box of Figure 4). By manipulating delegation relationships among context objects, behaviour can be adapted on the fly.

The context graph can be seen as a reification of the physical and logical environment in which the system is currently running. Each individual context object represents one part of such environment, and is generally domain-specific. In Figure 4 for instance, the `@telephony` context object has a number of proto-

⁶ Recall Figure 2 in Section 4.

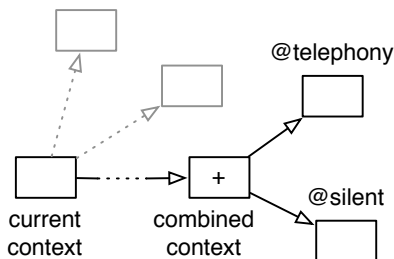


Figure 5: Adapted context graph.

types and method definitions that are about telephony. Another example is an *acoustics* context that contains functionality related to the noise or silence level of the surrounding environment. The behaviour of the system can be different if it is being used in a library, factory, on the street, and so on. In particular, the behaviour of the `advertise` method used by `receive` can be adapted to the current acoustic level. As explained in Section 2, the default implementation plays a ringtone through the phone speaker. However, behaviour that is better adapted to a silent environment can be defined as follows:

```
(with-contexts (@telephony @silent)
  (defmethod advertise ((call @phone-call) (phone @phone))
    (format t "Activating phone vibrator~%")))
```

This second version of `advertise` activates the phone vibrator, without producing sound. Note that this method is specialised on two context objects at the same time, namely `@telephony` and `@silent`, rather than only `@telephony` as the default version. This is an example of a *context combination*. Context combinations are context objects of their own, representing the combination as a whole. Behaviour that is specific to the particular combination can be defined as illustrated previously; other behaviour not specific to the combination is delegated to the constituent subcontexts, thanks to suitable delegation links as illustrated in Figure 5.

When the `@silent` context is activated (for example, if the system detects that a library has been entered), it will be combined with the currently active contexts. Delegation links among combined contexts are automatically maintained by the system, so that more specific combinations delegate to less specific ones. The current context object constitutes the most specific combination, whereas basic (non-combined) context objects such as `@telephony` and `@silent` are the least specific.

This concludes the explanation of the basic mechanism provided by *AmOS* to reify the dynamically changing context and adapt system behaviour accordingly. A more elaborate example and discussion of some of the issues related to

concurrent manipulation of the context graph is available in our previous paper on Ambience [10].

7 Discussion and Related Work

AmOS is a very dynamic computation model. It features dynamic dispatch⁷, dynamic inheritance, dynamic typing, and dynamic method scoping. One might very well wonder if such level of dynamism remains manageable. Although the answer is affirmative for small-scale scenarios, we still need to gather experience with larger case studies to assess the usefulness of the model in complex systems.

AmOS was initially inspired on Self [17] and Cecil [2], but later on adopted the similar, albeit more flexible, Prototypes with Multiple Dispatch (PMD) model [15]. Although the authors of PMD are well aware of the potential of subjective dispatch [15], it again faded into oblivion as happened with the Self extension Us [16]. We know of only one example showing the potential of subjective dispatch in the PMD model. *AmOS* can be seen as a version of PMD that boosts subjective dispatch, making it as fundamental to the model as prototypes and multimethods.

Soon after adopting the PMD model we became aware of ContextL [6], a class-based cousin of *AmOS*, which also exploits a sort of dynamic scoping mechanism to achieve behaviour adaptation. ContextL —an extension of CLOS— not only shares the similar goal of having behaviour depend on the context, but also a similar approach, by using an implicit argument that influences method dispatch. There are, however, two important differences.

Firstly, in ContextL there is one layer configuration (analogous to the context graph of *AmOS*) per thread. Threads cannot modify each other’s layer configurations. Whereas thread locality ensures non-interference with other threads, such interference is sometimes useful. In *AmOS*, there is a unique context graph that is shared by all threads; a context manager running in its own thread is in charge of updating the context graph in real time so that it matches the physical and logical environment as closely as possible, and all threads see such changes. Both approaches have their advantages and disadvantages. In *AmOS* the concurrent modification of the shared context graph can give rise to inconsistent behaviour [10]. In ContextL, the layer configuration must be adapted in the current thread, implying that context-switching constructs like `with-active-layers` and `ensure-active-layer` must be scattered throughout application code.

Secondly, *AmOS* is meant to be a model where all methods are dynamically adaptable. Having a distinction between adaptable and non-adaptable methods is analogous to having the `virtual` keyword in C++ for the declaration of dynamically bound methods. Foreseeing and fixing adaptability points

⁷ This synonym of multiple dispatch emphasises the fact that behaviour selection depends on the *dynamic* value of *all* arguments, rather than only one or none.

is limiting, as Java corroborates by having all methods be virtual. ContextL, in symbiosis with CLOS, does offer the possibility of defining all methods with `define-layered-method`, but does not advocate this choice as default option.

We have not made performance measurements yet. However, given that message sends are fully reflective,⁸ and there is no caching mechanism in place yet, chances are that our current implementation of *AmOS* does not match the speed of mature CLOS implementations and of CLOS extensions such as ContextL.

8 Conclusions and Future Work

Applications for Ambient Intelligence and Context-Oriented Programming require dynamic adaptation of behaviour according to the current physical and logical context in which the system is running. We have developed the Ambient Object System (*AmOS*), a simple yet flexible and expressive object model that aims at meeting the requirements of context adaptability. A few core concepts suffice to fully reify fundamental abstractions such as activations, closures and methods, and more innovative abstractions such as contexts and behaviour dependency on contexts.

We have fruitfully utilised Common Lisp for rapid prototyping and concept proof testing of *AmOS*. A good interactive development environment⁹ and the use of agile techniques such as unit testing have proved invaluable. We particularly took advantage of Common Lisp's powerful macro system to make the code look more lispy, with elaborate mechanisms being triggered under the surface. For instance, some variable accesses are actually symbol macros that expand to message sends.

In designing *AmOS* we have been mindful of future extensions to add concurrency and distribution. In particular, we are planning to extend *AmOS* with actor-based concurrency and dataflow synchronisation by means of asynchronous messages and futures.¹⁰ Regarding security, we need to assess the appropriateness of contexts (dynamic method scopes) as a simple visibility mechanism.

9 Acknowledgements

This work has been supported by the ICT Impulse Programme of the Institute for the encouragement of Scientific Research and Innovation of Brussels (ISRIB), and by the Interuniversity Attraction Poles (IAP) Programme of the Belgian State, Belgian Science Policy.

⁸ This means that the `(send selector arguments)` meta-method is executed for every message, bringing the advantages of meta-programming in our exploration of language semantics, to the detriment of performance.

⁹ SLIME, <http://common-lisp.net/project/slime/>.

¹⁰ This requires first-class messages, which we have not incorporated in *AmOS* yet.

References

1. Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A monotonic superclass linearization for dylan. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 69–82, New York, NY, USA, 1996. ACM Press.
2. Craig Chambers. Object-oriented multi-methods in cecil. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pages 33–56. Springer-Verlag, 1992.
3. Craig Chambers, David Ungar, and E. Lee. An efficient implementation of self, a dynamically-typed object-oriented language based on prototypes. *SIGPLAN Notices*, 24(10):49–70, 1989.
4. Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. Multi-Java: Design rationale, compiler implementation, and applications. *Transactions on Programming Languages and Systems (TOPLAS)*, 28(3), May 2006.
5. Pascal Costanza. Dynamically scoped functions as the essence of aop. *SIGPLAN Notices*, 38(8):29–36, 2003.
6. Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: an overview of ContextL. In *Dynamic Languages Symposium (DLS)*, pages 1–10. ACM Press, October 2005. Co-located with OOPSLA’05.
7. K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten, and J-C. Burgelman. Scenarios for ambient intelligence in 2010. Technical report, EC Information Society Technologies Advisory Group (ISTAG), 2001.
8. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.
9. M.L. Gassanenko. Context-oriented programming. In *euroForth’98*, April 1998.
10. Sebastián González, Kim Mens, and Patrick Heymans. Highly dynamic behaviour adaptability through prototypes with subjective multimethods. In *Dynamic Languages Symposium (DLS)*, pages 77–88, New York, NY, USA, October 2007. ACM Press. Co-located with OOPSLA’07.
11. Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology (JOT)*, March-April 2008. *To appear*.
12. Roger Keays and Andry Rakotonirainy. Context-oriented programming. In *MobiDe ’03: Proceedings of the 3rd ACM international workshop on Data Engineering for Wireless and Mobile Access*, pages 9–16, New York, NY, USA, 2003. ACM Press.
13. Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In Norman Meyrowitz, editor, *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 21, pages 214–223. ACM Press, 1986.
14. James Noble, Antero Taivalsaari, and Ivan Moore, editors. *Prototype-Based Programming: Concepts, Languages and Applications*. Springer-Verlag, 1999.
15. Lee Salzman and Jonathan Aldrich. Prototypes with multiple dispatch: An expressive and dynamic object model. In Andrew P. Black, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, LNCS 3586, pages 312–336. Springer-Verlag, 2005.
16. Randall B. Smith and David Ungar. A simple and unifying approach to subjective objects. *Theory and Practice of Object Systems (TAPOS)*, 2(3):161–178, 1996.
17. David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 227–242. ACM Press, 1987.

Visual Programming in PWGL

Mikael Laurson
(CMT, Sibelius Academy
laurson@siba.fi)

Mika Kuuskankare
(CMT, Sibelius Academy
mkuuskan@siba.fi)

Abstract: This paper gives an overview of how boxes are created in PWGL. PWGL is a visual language based on Common Lisp, CLOS and OpenGL. PWGL boxes can be categorized as follows. Simple boxes define the basic interface between PWGL and its base-languages Common Lisp and CLOS. Visual editors constitute another important subcategory of PWGL boxes. More complex boxes can be used to create PWGL applications ranging from simple ones to complex embedded boxes that can contain several editors and other types of input-boxes. We discuss the components of a PWGL box, how boxes are constructed and give some remarks on how to define the layout of a PWGL box.

Key Words: visual programming, computer-assisted composition, representation of musical structures

Category: D.1, D.1.7, I.2.5, J.5

1 Introduction

PWGL [Laurson and Kuuskankare 2006] is a visual language based on Common Lisp and CLOS with a strong emphasis on music related problems. PWGL is programmed with LispWorks (www.lispworks.com) ANSI Common Lisp [Steele 1990] that is source-code compatible across several different operating systems, such as OS X, Windows, and Linux. The graphics part of the system has been realized in OpenGL [Woo et al. 1999]. OpenGL offers several advantages such as multi-platform support, hardware acceleration, floating-point graphics and sophisticated 2D and 3D-graphics. Thus the PWGL system offers new potential that can be used to design more refined visual systems. PWGL is a free software and both Macintosh OS X and Windows XP versions can be loaded from our home page: www.siba.fi/pwgl.

PWGL is, along with OpenMusic [Assayag et al. 1999], a successor of PatchWork [Laurson 1996] and thus continues a long tradition of various concepts and tools that have proven to be useful for a large user-base consisting of composers, music theorists, and researchers. Typically this audience does not consist of professional programmers. Thus the strong visual approach provided by the PatchWork tradition gives an interesting alternative to learn and study programming. Other visual systems that are aimed for non-programmers include

for instance Alice [Kelleher and Pausch 2007]. Alice is, however, more entertainment oriented and it allows students to learn fundamental programming concepts in the context of creating animated movies and simple video games. PWGL, by contrast, can be considered as an expert system with specialized tools that aim to facilitate musical problem solving.

Among the strengths of the visual languages is the fact that the user can build the programming logic graphically by selecting different modules (usually boxes) and by making connections between these. The result resembles a flow chart that is often easier to read, modify and understand than text-based programs. Also, visual languages are simpler to master as they usually have a more uniform and intuitive syntax than text-based languages. Essentially, the syntax consists of making the right connections between the right boxes. As PWGL is specialized in solving complex musical problems the visual aspect of our system is even more important. Musical objects, such as scores, can be recognized and modified efficiently, program structures can be parsed, understood and manipulated more easily. When displaying musical material visually on-screen, we can combine the benefits of traditional score representation with the novel and dynamic possibilities of the computer.

PWGL is a multi-window system. A PWGL window is called a patch. A patch, in turn, contains boxes and connections. In the simplest case a box is a visual equivalent to a Lisp function or method. It has a number of input-boxes - containing typically constants such as numbers or lists - and one or several outputs. When evaluated a box reads its inputs, calls a function or method associated to it and finally returns a result. Connections are used to define relations between boxes. An output of a box can be connected to an input-box of another box. Thus the system works in a similar fashion than Lisp where function calls can have as arguments either constants or functions calls.

PWGL has an object-oriented graphical user interface (GUI) that is based on direct manipulation [Cooper 1995]. The underlying idea behind the PWGL GUI is to allow the user to manipulate the information contained by the patch and its various graphical editors as straightforwardly as possible. As a general guideline, any object of any complexity can be edited directly, and all editing operations provide synchronized visual feedback for the user.

Figure 1 shows a relatively complex PWGL example with musical material situated in various musical editors. It is important to note that in addition to an algorithmic approach all editors can be edited by hand.

In the following we will concentrate on the most important visual entity in the system: the PWGL box. A PWGL box consists of a number of input-boxes. PWGL has a library of predefined input-boxes which typically handle numbers, lists and popup-menus. PWGL has also an important subgroup of input-boxes that are associated to editor-windows. These editor-windows contain complex

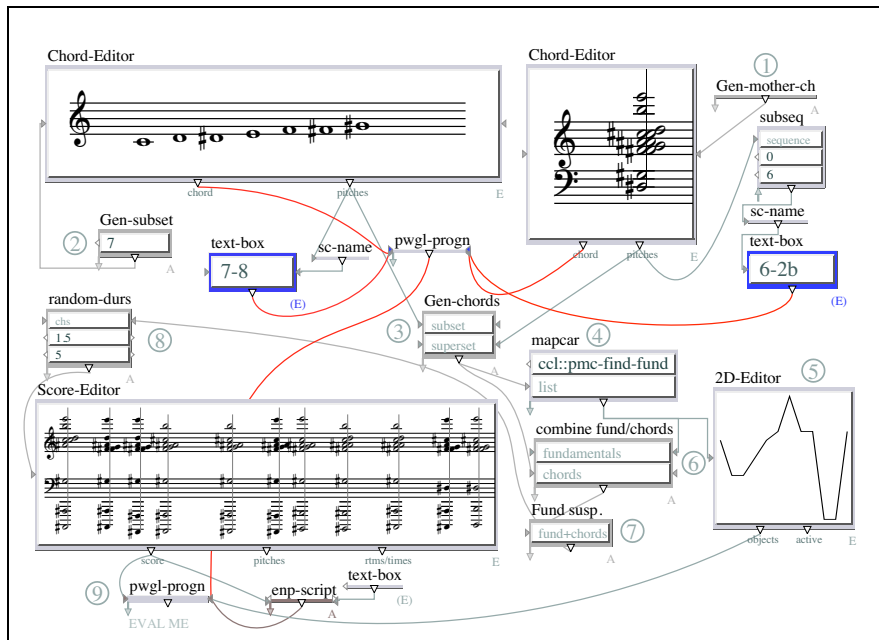


Figure 1: A musical search problem where a pitch-class subset is mapped to a larger 12-tone symmetric ‘mother-chord’. This results (see the lowest score editor) in a sequence of chords that all contain the given subset. The example is also enriched by adding octaves to all chord formations. These octaves form in turn a suspension-release pattern.

objects, such as scores, chords, break-point functions, bezier functions and sound samples. These input-boxes can be opened and inspected or edited by the user.

PWGL offers several ways to construct boxes ranging from completely automatic to methods that allow to specify the exact type of input-boxes, default-values and layout options. All PWGL boxes can be resized both vertically and horizontally. This option adds new requirements to our system. It has to deal with boxes that are not just simple fixed-sized rectangles. Instead, boxes have to behave in a coherent manner after the size of the box has been changed.

When compared to other music related visual languages - such as OpenMusic, Max/MSP (distributed by Cycling ’74), and PD [Puckette 1996]- PWGL with its OpenGL-based graphical environment has many features and facilities that are unique. Several of these aspects have been already mentioned above, such as direct editing, rich set of input-box types, resizable boxes, user-definable layout options, and professional quality music notation with sophisticated GUI.

The rest of the paper is organized as follows. First, we briefly give a general

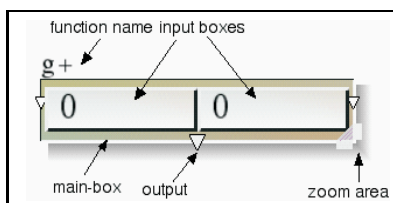


Figure 2: PWGL box components.

overview of a PWGL box and enumerate the main features that are shared by all PWGL boxes. The next section deals with the creation of boxes. We start with simple Lisp definitions and go over to more complex options that allow to define a box in a more precise manner. Then we discuss a method which allows the user to define a box in great detail. The user can specify using various layout options how the input-boxes will be distributed within the box and how the input-boxes will respond when the box is being resized. We end with two complex case studies. First, we show how the user can define a complex embedded box. Finally, we give a complete box example that aims to demonstrate how various input-boxes can interact with each other.

2 PWGL Boxes

2.1 Box Components

This section discusses the main components of a PWGL box (Figure 2). A box consists of a main-box and a number of input-boxes. The function-name is given above the top-left corner of the main-box. The bottom-right corner contains a zoom area allowing the user to modify the size and shape of the box. The user can evaluate the selected box by typing the character 'v'. If there are several outputs the user can select the desired one directly by clicking the output triangle. If no output is selected then the left-most output is used.

When the user moves the mouse above a box the cursor changes its shape depending on which part of the box the mouse is currently located in (Figure 3). Figure 3 shows also the actions that will occur if the user clicks the mouse and starts to drag it.

2.2 Lambda-list Keyword Support

PWGL supports automatically the most commonly used Common Lisp lambda-list keywords (i.e., &optional, &rest and &key, [Steele 1990]). In the simplest case where a Lisp lambda-list contains only the required arguments - Figures 2 and 3

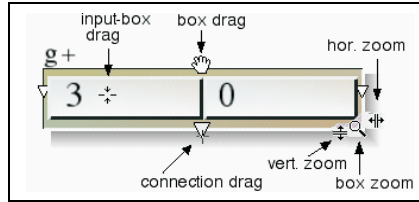


Figure 3: Various cursor shapes and the associated actions of a PWGL box.

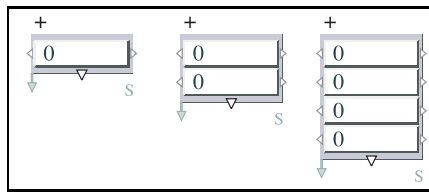


Figure 4: Extendable boxes of type `&rest`.

show an example of such a box having 2 required inputs - the system generates automatically one input-box for each required argument. In the case of keyword arguments the box is extendable and contains a downward pointing arrow at the bottom-left corner of the main-box. Figure 4 shows a box that represents the Lisp function `'+'` that has in the argument list the keyword `&rest`, (i.e., the box can have an arbitrary number of input-boxes). Figure 4 shows instances of the `'+'` box having 1, 2 and 4 input-boxes:

Figure 5, in turn, gives a more complex example using the Lisp function `'position'` that has 2 required arguments and 6 `&key` arguments. The left-most box contains only the required arguments while the one to the right has one `&key` argument. The `&key` arguments extend the box with 2 input-boxes at a time where the first one is a popup-menu indicating the keyword (here `':key'`) and the latter one giving the value for this keyword (here `'first'`):

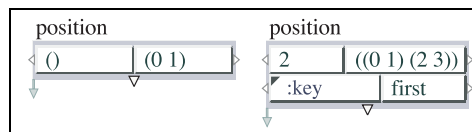


Figure 5: Extendable boxes of type `&key`.

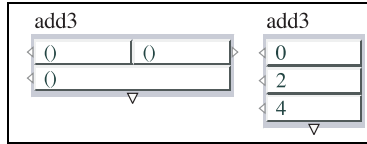


Figure 6: Two box variants of ‘add3’.

3 Box Creation

There are three different schemes that can be used to generate PWGL boxes. In the first one the user simply defines a Lisp function using the standard macro `defun`. The system generates automatically the corresponding PWGL box using the knowledge of the underlying Lisp system. For instance, let us assume the following Lisp function:

```
(defun add3 (a b c)
  "simple add"
  (+ a b c))
```

The function can be converted automatically to a box by typing the name of the function in a dialog box (see the resulting box to the left in Figure 6). The second and somewhat similar approach to create boxes consists of using a PWGL macro called `PWGLDef`. The most important difference between `defun` and `PWGLDef` is that in the latter case the macro creates internally a generic function. Furthermore, the user can specify the input-box type and the default value for each argument. Furthermore, `PWGLDef` accepts a list of extra keyword/value pairs that allow to define the outlook and behavior of a box in more detail. Let us assume that we would like to change the previous box definition in two ways. First, we give default arguments for each input. Second, we change the default grouping so that the box would consist of a column of 3 input-boxes. These changes are achieved by the following definition (the corresponding box can be found to the right in Figure 6):

```
(PWGLDef add3 ((a 0)(b 2)(c 4))
  "simple add"
  (:groupings '(1 1 1))
  (+ a b c))
```

The third method to create boxes consists of using the ‘`mk-box-function`’ method. Here the user can specify the required input-box types, default values, outputs and layout-options in the most detailed form. We give next the code to create a box with 3 editor input-boxes each having an initial state, 1 horizontal slider and 3 outputs. The resulting box can be found in Figure 7.

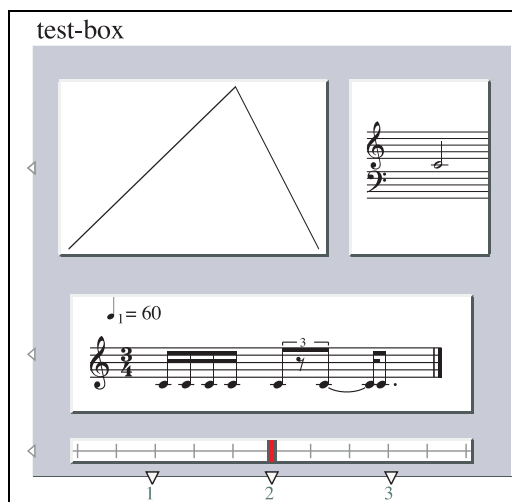


Figure 7: A box containing a 2D-editor, chord-editor, score-editor, slider and 3 outputs.

```
(defmethod mk-box-function ((self (eql test-box)) x y)
  (mk-PWGL-box
   PWGL-box self "test-box" x y 1.0 0.9
   (list (mk-2D-subview :application-window
                       (mk-2D-application-window
                        :2D-subviews (list (mk-bpf '(0 2 3) '(0 1 0)))))
         (mk-default-chord-subview)
         (mk-score-subview
          :application-window
          (make-enp-application-window '((((1 (1 1 1 1))(1 (1 -1 1))(1 (1.0 3)))))))
         (mk-slider-subview :value 50 :minval 0 :maxval 100 :grid t :horizontal t))
   :proportional-coordinates
   '((1/15 1/12 5/9 4/10) (8/12 1/12 2/7 4/10)
     (1/12 7/12 10/12 4/15) (1/12 11/12 10/12 1/20))
   :outputs (list "1" "2" "3"))
```

4 Box Layout

As all PWGL boxes can be resized special attention must be given to layout options as the input-boxes of a box cannot be positioned simply by using static x-y coordinates. The key point here is to use proportional values instead of fixed coordinate values. Similar kind of dynamically resizable objects - typically dialog windows - can also be found for instance in Mac OSX and in some programming environments such as the CAPI system [LispWorks: CAPI User Guide].

When defining the layout of a box PWGL uses two sets of keywords:

(1) :groupings, :x-proportions, :y-proportions

or

(2) :proportional-coordinates

In (1) the data lists :y-proportions and :x-proportions give proportional delta-values (the delta-values are scaled so that their sum equals 1.0 in order to guarantee that the subviews will always be inside the main-box). The :groupings keyword is a list of values where each value gives the number of subviews for each row (thus a list (3 3) groups 6 subviews into two rows, where each row has 3 subviews). :y-proportions is a list of proportional delta-values defining the height of each row. If not given then all rows have equal height. :x-proportions is a list of lists of proportional delta-values. Each sublist defines the internal x proportions of the respective row of boxes. If not given then each subview within a row has equal width.

Option (1) is often easier to use than option (2) as it requires only a small amount of data to be functional. For instance the second version of the 'add3' box example (see Figure 6) required only the groupings list (1 1 1) to define the layout of a box where the input-boxes form a column. There are, however, some restrictions. There can be no overlaps, no holes between input-boxes (holes can though be simulated with special subviews) and subviews are always aligned in horizontal direction. In option (2) - using :proportional-coordinates - the data lists give proportional coordinates for each subview in the form: ((x1 y1 w1 h1)) ... (xN yN wN hN)) where each sublist defines the proportional x- and y-position and proportional width and height of the respective subview (note: these values are not scaled). While the :proportional-coordinates option requires often more data than option (1), it has some advantages. Subviews can be freely distributed, they can be positioned outside the main-box and overlaps can occur. Figure 7 shows an example how to use the :proportional-coordinates option to define a box layout.

Sometimes the use of pure proportional delta-values or coordinates leads to undesired results. A typical example is for instance a box containing sliders that function as scroll-bars. In this case it is probably more desirable if scroll-bars have fixed size in one dimension while other subviews are resized dynamically as before. This behavior can be achieved by using a mixed form of delta-values or coordinates. Whenever the system encounters a list consisting of the keyword :fix and a value, then the value is considered to be fixed and not proportional.

Let us assume a box consisting of two rows of subviews. The first row from the top contains a 2D-editor (a PWGL 2D-editor is a short for '2-Dimensional editor', i.e. an editor containing objects that can be displayed in 2 dimensions) and a vertical scroll-bar. The second row, in turn, has a horizontal scroll-bar and a small button-subview (see the box to the left in Figure 8). If we use the following layout data:

```
:groupings '(2 2)
:x-proportions '((20 1) (20 1))
:y-proportions '(20 1)
```

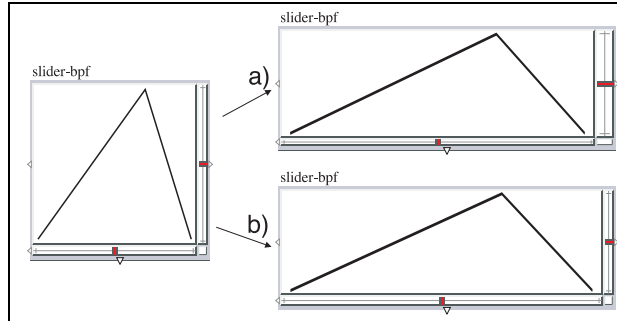


Figure 8: A PWGL box and two resized box versions with different layout data: a) pure proportional delta-values, b) mixed delta-values.

we get a box - after resizing it horizontally - where the width of the vertical scroll bar differs from the height of the horizontal scroll bar (Figure 8 to the right, upper box). If, however we use the following mixed form of layout data (note the expressions starting with the keyword `:fix`):

```
:groupings '(2 2)
:x-proportions '((20 (:fix 0.03)) (20 (:fix 0.03)))
:y-proportions '(20 (:fix 0.03))
```

the width of the vertical scroll-bar and the height of the horizontal scroll-bar are always fixed to 0.03 units (see the lower box to the right of Figure 8).

5 Recursive Boxes

A PWGL box can also be recursive, i.e., it can contain instances of itself. This property allows to combine features described above into one complex box. Figure 9 shows a main box containing 3 sub-boxes. Each sub-box can have its own background color, subviews and layout. This scheme is very useful as it permits to define a library of box components (similar to the library of basic input-boxes) that can be used as building blocks when constructing even more complex boxes.

In the following we give the Lisp code that was used to create the box in Figure 9. We start by defining 3 functions. The first one, 'mk-test-bx1', creates a box with 1 2D-editor, 2 score-editors, and 1 2D-editor. The second box, 'mk-test-bx2', consists of 1 slider-bank, 2 score-editors, and 1 2D-editor. The third one, 'mk-test-bx3', has 2 score-editor boxes. Finally, the function 'mk-recursive-bx' uses these functions to build the final box. It groups - `:groupings '(2 1)` - the two first boxes in the first row, while the third box is situated in the second row (see Figure 9).

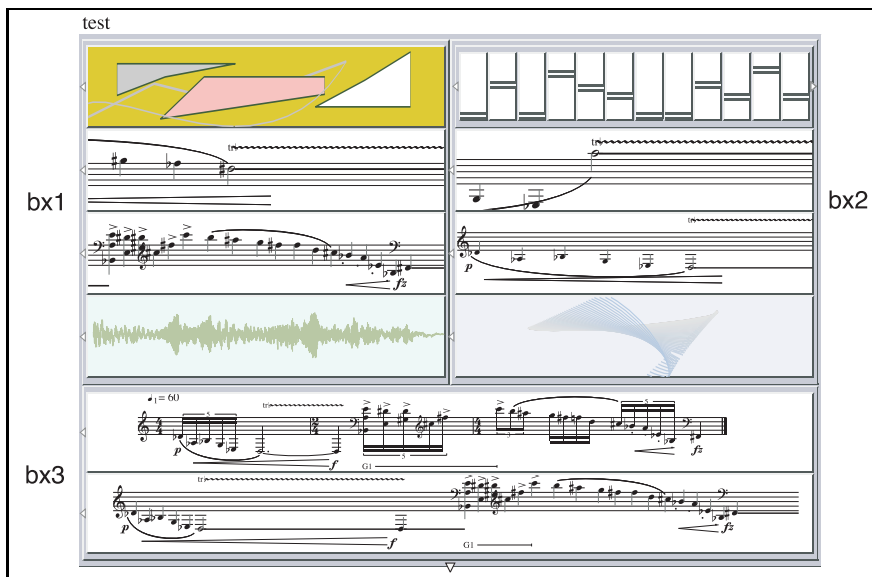


Figure 9: A complex recursive box.

```
(defun mk-test-bx1 (&optional (x 0) (y 0))
  (mk-PWGL-box 'PWGL-box 'bx1 "bx1" x y 1.0 0.9
    (list (mk-2D-subview :application-window (mk-2D-application-window))
          (mk-score-subview :application-window (make-enp-application-window '((((())))))
          (mk-score-subview :application-window (make-enp-application-window '((((())))))
          (mk-2D-subview :application-window (mk-2D-application-window)))
          :groupings '(1 1 1 1)))

(defun mk-test-bx2 (&optional (x 0) (y 0))
  (mk-PWGL-box 'PWGL-box 'bx2 "bx2" x y 1.0 0.9
    (list (mk-slider-bank (loop for i from 1 to 12 collect (format () "f~A" i)) ()) :display ())
          (mk-score-subview :application-window (make-enp-application-window '((((())))))
          (mk-score-subview :application-window (make-enp-application-window '((((())))))
          (mk-2D-subview :application-window (mk-2D-application-window)))
          :groupings '(1 1 1 1)))

(defun mk-test-bx3 (&optional (x 0) (y 0))
  (mk-PWGL-box 'PWGL-box 'bx3 "bx3" x y 1.0 0.9
    (list (mk-score-subview :application-window (make-enp-application-window '((((())))))
          (mk-score-subview :application-window (make-enp-application-window '((((())))))
          :groupings '(1 1)))

(defun mk-recursive-bx (&optional (x 0) (y 0))
  (mk-PWGL-box 'PWGL-box 'recursive "test" x y 1.0 0.9
    (list (mk-test-bx1)
          (mk-test-bx2)
          (mk-test-bx3))
          :groupings '(2 1)
          :y-proportions '(2 1)))
```


6 Subclassing a PWGL box

Until now our focus has been mostly in the visual appearance of a box. In order to create a fully functional subclass of the main box class, called ‘PWGL-box’, the user has typically to consider the following steps:

- (1) define a new generic function
- (2) define a subclass of ‘PWGL-box’
- (3) define a ‘patch-value’ method for the new class
- (4) define a ‘mk-box-function’ or a ‘PWGLDef’ method for the new box
- (5) if needed, define some Lisp code for updates, user interaction, etc.

Next we describe a case study where we create a subclass of ‘PWGL-box’. The visual outlook – the first row of input-boxes consisting of 2 2D-editors (‘2DA’ and ‘2DB’), the second row of 1 2D-editor (‘2DC’), and the third row of 1 slider – can be seen in Figure 10. The box has also 3 outputs labelled ‘resbpf’, ‘bpf1’, and ‘bpf2’.

In the following code fragment we first define a generic function (1) and then a new box class (2). In (3) we define the ‘patch-value’ method for the new box class (‘patch-value’ is the generic function that is called when a box evaluates itself). Here we call a new ‘multiout-patch-value’ method that has 3 definitions, one for each output. Each ‘multiout-patch-value’ method returns one break-point function contained in the box depending on the output label. Thus the user can access each break-point function separately in a patch, if needed.

In (4) we specify the input-boxes, box layout, and so on, as has already been explained in the previous sections of this article. An interesting detail is that we can name each input-box (using the ‘:pwgl-nick-name’ keyword argument), so that they can be accessed more easily in the code. The access is achieved with the function ‘find-by-nick-name’, that was already used in the ‘multiout-patch-value’ methods. An important addition is finally the ‘:pwgl-action-function’ keyword, that is used here to define an update function for the slider.

Finally, in (5) we find a special ‘action-function’, called ‘update-interpol-bpf’, that is called each time the user moves the handle part of the slider. Here again we use the ‘find-by-nick-name’ function to access various parts of the main box. The action-function reads the 2 break-point functions (‘bpf1’ and ‘bpf2’) from the two first 2 2D-editors (‘2DA’ and ‘2DB’) and interpolates the x-values and y-values of them according to the position of the slider handle (left means only ‘bpf1’, right means only ‘bpf2’, middle means 50% of ‘bpf1’ and 50% of ‘bpf2’, and so on). This results in a new break-point function that is stored in ‘2DC’.

```
;;------(1)-----  
(defgeneric interpol-bpfs ()  
  (:documentation  
   "Interpolate two bpfs ('bpf1' and 'bpf2') in the first row with a slider,  
   the result is shown as a bpf ('resbpf') in the second row.")  
  )  
;;------(2)-----
```

```

(defclass PWGL-interpol-test-box (PWGL-box) ())

;;------(3)-----
(defmethod patch-value :around ((self PWGL-interpol-test-box) outbox)
  (multiout-patch-value self (read-from-string (format () "~A" (box-string outbox)))))

(defmethod multiout-patch-value ((self PWGL-interpol-test-box) (outnum (eql :bpf1)))
  (first (2D-editor-objects (application-window (find-by-nick-name self :2DA)))))

(defmethod multiout-patch-value ((self PWGL-interpol-test-box) (outnum (eql :bpf2)))
  (first (2D-editor-objects (application-window (find-by-nick-name self :2DB)))))

(defmethod multiout-patch-value ((self PWGL-interpol-test-box) (outnum (eql :resbpf)))
  (first (2D-editor-objects (application-window (find-by-nick-name self :2DC)))))

;;------(4)-----
(defmethod mk-box-function ((self (eql 'interpol-bpfs)) x y)
  (mk-PWGL-box
   'PWGL-interpol-test-box self "Interpol bpfs" x y 0.5 0.5
   (list
    (mk-2D-subview :application-window (mk-2D-application-window)
                  :pwgl-nick-name :2DA )
    (mk-2D-subview :application-window (mk-2D-application-window)
                  :pwgl-nick-name :2DB)
    (mk-2D-subview :application-window (mk-2D-application-window)
                  :pwgl-nick-name :2DC)
    (mk-slider-subview :value 0 :minval 0 :maxval 100 :horizontal t :grid t :grid-step 10
                      :pwgl-nick-name :interpol-slider :pwgl-action-function 'update-interpol-bpf))
   :groupings '(2 1 1)
   :x-proportions '((5 5) (1) (1))
   :y-proportions '(5 10 (:fix 0.03))
   :outputs (list "resbpf" "bpf1" "bpf2"))))

;;------(5)-----
(defun update-interpol-bpf (slider)
  (let* ((container (pwgl-view-container slider))
         (bpf1 (first (2D-editor-objects (application-window (find-by-nick-name container :2DA)))))
         (bpf2 (first (2D-editor-objects (application-window (find-by-nick-name container :2DB)))))
         (bpf3-win (application-window (find-by-nick-name container :2DC))))
    (when (and bpf1 bpf2 bpf3-win)
      (set-2D-editor-objects bpf3-win
        (list (mk-bpf (list-interpolation (x-points bpf1) (x-points bpf2) (curval slider) 101 1.0)
                    (list-interpolation (y-points bpf1) (y-points bpf2) (curval slider) 101 1.0))))
      (redraw-pwgl-window (pwgl-win container)))))

```

7 Conclusions and Future Work

This paper gave a survey of OpenGL-based visual PWGL boxes. We first presented the main components of a box. After this we discussed different options how to construct boxes and gave some ideas of available layout schemes. We gave also code examples to realize two complex case studies demonstrating some of the more advanced possibilities in box design in PWGL.

Although the system is already functional it can be extended and improved in several ways. One idea is to add more layout options that for example would allow to control in more detail how boxes respond to resize operations. The current system could easily be extended to support other types of mixed delta-values or proportional coordinates.

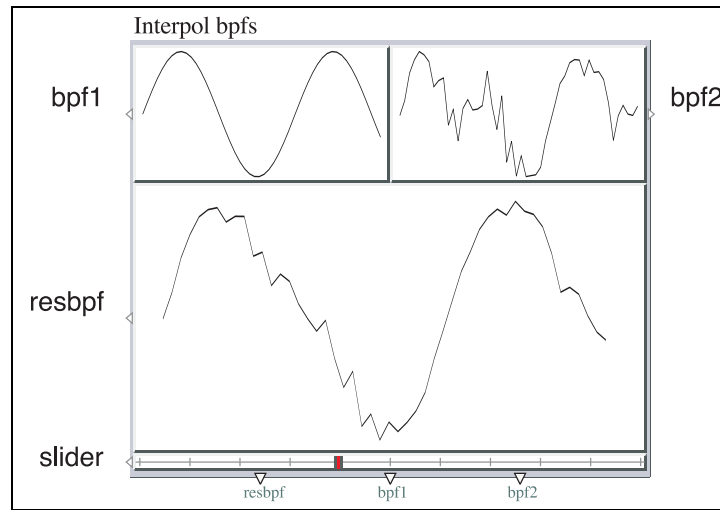


Figure 10: A box that interpolates two break-point functions ('bpf1' and 'bpf2') with a slider. The resulting break-point function ('resbpf') is shown in the second row.

8 ACKNOWLEDGEMENTS

The work of Mikael Laurson and Mika Kuuskankare has been supported by the Academy of Finland (SA 105557 and SA 114116).

References

- [Assayag et al. 1999] Assayag G., Rueda C., Laurson M., Agon C., and Delerue O.: "Computer Assisted Composition at IRCAM: From PatchWork to OpenMusic"; *Computer Music Journal*, vol. 23, pp. 5972, Fall 1999.
- [Cooper 1995] Cooper A.: "About Face. The Essentials of User Interface Design"; Foster City, CA: IDG Books, 1995.
- [Kelleher and Pausch 2007] Kelleher, C. and Pausch, R.: "Using Storytelling to Motivate Programming"; *Communications of the ACM*, vol. 50 no. 7, pp. 58-64, July 2007.
- [Laurson 1996] Laurson M.: "A Visual Programming Language and Some Musical Applications"; Doctoral dissertation, Sibelius Academy, Helsinki, Finland, 1996.
- [Laurson and Kuuskankare 2006] Laurson M. and Kuuskankare M.: "Recent Trends in PWGL; International Computer Music Conference, (New Orleans, USA), pp. 258261, 2006.
- [LispWorks: CAPI User Guide] LispWorks: CAPI User Guide: <http://www.lispworks.com/>.
- [Puckette 1996] Puckette M.S.: "Pure Data; International Computer Music Conference, (San Francisco, USA), pp. 269-272, 1996.
- [Steele 1990] Steele G. L. JR.. "COMMON LISP THE LANGUAGE"; Digital Press, 2nd edition, Massachusetts, USA, 1990.

[Woo et al. 1999] Woo M., Neider J. , Davis T., and Shreiner D.: “OpenGL Programming Guide”; Addison Wesley, 3rd edition, Massachusetts, USA, 1999.

UCL-GLORP—An ORM for Common Lisp

António Menezes Leitão

aml@gia.is.utl.pt

INESC-ID/Technical University of Lisbon

Rua Alves Redol, n. 9, Lisboa, Portugal

Abstract: UCL-GLORP is a Common Lisp implementation and extension of GLORP (**Generic Lightweight Object-Relational Persistence**), an Object-Relational Mapper for the Smalltalk language. UCL-GLORP is now a mature framework that largely extends GLORP and that takes advantage of some of Common Lisp unique features. This paper illustrates UCL-GLORP and discusses some of the challenges that we faced in order to find suitable replacements, in Common Lisp, for some of the more esoteric features of Smalltalk that were explored by GLORP.

Key Words: Object-relational mapping, Common Lisp, Smalltalk

Category: D.1.5, D.2.2, D.3.3, H.2

1 Introduction

A large fraction of modern applications need to store information in some persistent form. Although object-oriented databases are much more trendy, relational databases are still the dominant technology for providing data persistence and, in many cases, they are also a requirement.

Being forced to store all data in a relational model doesn't mean that the application can not be programmed in a modern object-oriented style. For all mainstream object-oriented languages there exist one or more Object-Relational Mappers (ORM) that can be programmed to transform data from an object-oriented model into a relational model.

Until very recently, the only ORM available for Common Lisp was CLSQL[11] but, unfortunately, it doesn't provide many of the important features identified by Fowler [2]:

- It doesn't properly implement the Identity Map pattern so it doesn't preserve the identity of loaded objects implying that an object has as many copies as the number of times it was loaded from the database. Besides the time and memory waste, this creates severe identity problems such as inconsistent updates to the "same" object.¹

¹ CLSQL implements a cache where objects are related to the queries that loaded them but a different query that happens to return some previously loaded object will not notice it.

- It doesn't implement the Unit of Work pattern, forcing the programmer to either manually save all updated objects or else to rely on CLSQL's automatic save mechanism that occurs on every slot update and that causes performance problems due to the amount of database calls.
- It doesn't implement the concept of Object Transaction, meaning that if a database transaction fails while updating some rows, the mapped objects in the application no longer reflect their last saved state. An Object Transaction provides the same purpose as a database transaction but on the object level, thus maintaining the consistency between them.
- It doesn't implement the Optimistic Offline Locking pattern that is based on the number of modified rows. This last functionality can easily be added to CLSQL (we did it) but it is harder to automatically consider it for the detection of concurrent updates and the necessary signaling of the corresponding object transaction failure.

As a result, CLSQL doesn't qualify as a proper ORM. Given the huge amount of effort that is required for developing an ORM from scratch, we decided to adopt a different strategy based on the translation of some already developed ORM from its original programming language to the Common Lisp language. After looking for a sufficiently developed ORM that was available with an adequate license, we end up selecting GLORP—the **Generic Lightweight Object-Relational Persistence**.

In the next section we will briefly highlight some of the more important characteristics of GLORP. Then, in section 3, we will discuss UCL-GLORP, our rewrite and extension of GLORP for the Common Lisp language. Section 4 will discuss the problems found and the solutions adopted and, finally, section 7 will present the conclusions.

2 GLORP

GLORP is an open-source object-relational mapping layer for Smalltalk running in several different implementations, including VisualWorks, VisualAge, Dolphin Smalltalk and Squeak. GLORP features a sophisticated mapping layer that uses a declarative approach to map classes to tables, instance variables to columns and references to foreign keys.

Besides being an ORM, GLORP is also a showcase for the principles and patterns that underlie all ORMs. In the next subsections, we will discuss some of those patterns.

2.1 Models and Mappings

GLORP depends on explicit mappings between objects and their database representations. These mappings operate over object models and database table models.

Each object model describes all the attributes of a specific type of object, in particular, all its relevant slots, their datatypes, their readers and writers, etc. The object model is fundamental because it usually contains much more information than what is generally available in a Smalltalk class definition.

Each table model describes all the attributes of a database table, including column names and types, primary keys, foreign keys, constraints, etc. Although the table model can model a legacy database schema, it is also possible to use it to automatically create the corresponding database schema.

Based on both the object models and the table models, several kinds of mappings can be established but two of them are the most used: object slots containing value objects [2] use *direct mappings*, i.e., they are mapped to the corresponding table columns; object slots containing reference objects are mapped to foreign keys, using *one-to-one*, *one-to-many* and *many-to-many* mappings. All these mappings are crucial to translate object operations to database operations.

2.2 Units of Work and Transactions

Instead of forcing the programmer to explicitly write code that, for each updated object, also updates the database, GLORP automatically computes the necessary database updates based on the objects that were loaded, created, modified or deleted during a *unit of work*. This not only simplifies the programmer's work but it is also important to allow reordering of the database updates so that all integrity constraints are satisfied.

Besides Units of Work, GLORP also provides transactions at the object level. This means that, for each object that is modified, a shallow copy is created that contains the previous values of the object slots so that, if necessary, each modified object can be restored to its previous state. This mechanism is important to provide consistency between the database and the application level. Whenever a database transaction aborts, the application program is notified and it can choose to also abort, undoing all object changes that were made during the unit of work.

GLORP contains many other features that are worth discussing but that are beyond the scope of this paper. We refer the reader to [7].

3 UCL-GLORP

Given the flexibility and sophistication of GLORP, it was tempting to use it as the basis for a Common Lisp ORM implementation. The plan was to first

semi-automatically translate GLORP from Smalltalk to Common Lisp and then to further develop it so that it could take advantage of the new implementation language. However, implementing and extending GLORP in Common Lisp was far from simple and required us to explore less well-known features of the Common Lisp language. At times, we had the feeling that we were “pushing the envelope” of Common Lisp far beyond its original design. We will postpone the discussion of the problems found until section 4 and we will now describe UCL-GLORP, the Common Lisp implementation of GLORP.

UCL-GLORP is an ORM for Lisp. Like GLORP, UCL-GLORP depends on class models and, given the variety of object systems available in the Lisp world, we designed it to be independent of any specific object system, as long as it is class-based. However, some of the more advanced features do depend on the Common Lisp Object System so, in this paper, we will restrict the discussion to the use of UCL-GLORP as an ORM for CLOS.

3.1 Models and Mappings

The first step to provide CLOS classes with relational persistence is to define the class models, the table models and the mappings between them. The most flexible approach is to manually specify that information, allowing complete freedom over table and column names, types, indexes and constraints. In many cases, however, there is a strong correlation between the CLOS classes and the database schema. For these cases, UCL-GLORP is capable of inferring models and mappings strictly from plain CLOS classes, as long as the Common Lisp implementation allows class introspection (e.g., using the CLOS MOP [5]). We will now demonstrate this capability by modeling in CLOS a small database to keep people names and their home address:

```
(defclass person ()
  ((name :type string :initarg :name :accessor name)
   (address :type address :initarg :address :accessor address)))

(defclass address ()
  ((street :initarg :street :type string :accessor street)
   (city :initarg :city :type string :accessor city)))
```

It is important to stress that the previous classes are plain CLOS classes: `defclass` was not shadowed and there are no metaclasses involved. However, we included with the slots information regarding their types and these type declarations allow UCL-GLORP to infer not only the types to use in the corresponding database columns but also the relationship between `person` and `address`.

The next step consists of selecting the intended database *platform* (e.g., postgres, oracle, mysql, etc), the intended database *accessor* (e.g., clsql, cl-rdbms,

etc), and, finally, the necessary database *login* information for accessing the database.²

All these steps can be done using the function `make-clos-session` that also creates a *session* for talking with the relational database:

```
(defparameter *session*
  (make-clos-session
   :classes '(person address)
   :username "foo" :password "bar" :database "baz"))
```

Just like GLORP, UCL-GLORP can use any legacy database model but can also automatically create the tables using the following expression:

```
(recreate-tables *session*)
```

This causes UCL-GLORP to issue the following SQL commands to the database:

```
CREATE TABLE person (oid serial NOT NULL,name text NULL,address int8 NULL,
  CONSTRAINT person_pk PRIMARY KEY (oid),
  CONSTRAINT person_uniq UNIQUE (oid))
CREATE TABLE address (oid serial NOT NULL,street text NULL,city text NULL,
  CONSTRAINT address_pk PRIMARY KEY (oid),
  CONSTRAINT address_uniq UNIQUE (oid))
ALTER TABLE person ADD CONSTRAINT person_add_to_address_oi_ref1
  FOREIGN KEY (address) REFERENCES address (oid)
```

Note that a *primary key oid (object id)* column was included on both tables and that a *foreign key address* was included in the table `person` so that each person row can reference its address row. These decisions were made automatically by UCL-GLORP but could have been overridden by the user.

3.2 Storing and Retrieving

Using the created session, it is now possible to give persistence to our objects. This is accomplished using a `with-unit-of-work` form that keeps track of all the manipulated objects during its dynamic scope. At the end, the unit of work computes the necessary changes to the database and starts a database transaction to persist those changes. Here is one example:

```
(with-session (*session*)
  (with-unit-of-work ()
    (db-persist
      (make-instance 'person
        :name "John Adams"
        :address (make-instance 'address
          :street "Park Avenue"
          :city "New York")))))
```

² In the following examples, we will use the postgres platform and the `clsq` accessor.

Note that persisting one object entails persisting all objects reachable from it. The generated SQL is the following:

```
BEGIN TRANSACTION
SELECT nextval('address_oid_seq') FROM pg_attribute LIMIT 1
SELECT nextval('person_oid_seq') FROM pg_attribute LIMIT 1
INSERT INTO address (oid,street,city) VALUES (1,'Park Avenue','New York')
INSERT INTO person (oid,name,address) VALUES (1,'John Adams',1)
COMMIT TRANSACTION
```

As is possible to see from the SQL log, UCL-GLORP assigns oids to the rows using database sequences and then inserts them in their respective tables.

It is now safe to shutdown the Common Lisp process. Upon restart, the persisted objects can be reloaded using the `db-read` function. This function accepts many options (some will be described later) but, for the moment, it is sufficient to say that it is possible to read just `:one` instance of the specified class or `:all` stored instances of that class. Here is one expression that returns the previously stored person:

```
(with-session (*session*)
  (let ((p (db-read :one 'person)))
    (describe p)))
```

The evaluation of the previous expression issues the following SQL statement:

```
SELECT t1.oid, t1.name, t1.address FROM person t1 LIMIT 1
```

and prints:

```
#<PERSON @ #x7352377a> is an instance of #<STANDARD-CLASS PERSON>:
The following slots have :INSTANCE allocation:
  NAME      "John Adams"
  ADDRESS   <unbound>
```

Note that the `address` slot is unbound. This is intended because UCL-GLORP uses *lazy loading* [2]: referenced objects are loaded only when needed.³ However, on the first attempt to access the currently unbound slot, UCL-GLORP will “resolve” it using another SQL statement:

```
SELECT t1.oid, t1.street, t1.city FROM address t1
WHERE (t1.oid = 1) LIMIT 1
```

The result is then used to build the appropriate `address` instance that is stored in the previously unbound slot so that future slot accesses behave as usual. Again, we should stress that this mechanism didn't require any special care from the programmer. All that was needed was to wrap the code in a `with-session` form.

³ This behavior can be customized by the programmer on a slot by slot basis.

3.3 Relations

In the previous example, each person references one address but we are not restricted to one-to-one relations. We can also have one-to-many and many-to-many relations. For example, let's suppose each person also has a vector of email addresses. This can be written using a `(vector email-address)` compound type specifier, as follows:

```
(defclass person ()
  ((name :type string :initarg :name :accessor name)
   (address :type address :initarg :address :accessor address)
   (email-addresses :type (vector email-address)
                    :initarg :email-addresses :accessor email-addresses)))

(defclass email-address ()
  ((username :initarg :username :type string :accessor username)
   (host :initarg :host :type string :accessor host)))
```

UCL-GLORP will use the `:type` option in the `email-addresses` slot to infer a one-to-many relation from `person` to `email-address`.⁴ This implies that UCL-GLORP will include a foreign key column in the table for email addresses that will point to the person that owns the email address, as is possible to see in the generated SQL for the table `email_address`:

```
CREATE TABLE email_address (oid serial NOT NULL,username text NULL,
  host text NULL,person_email_addresses int8 NULL,
  CONSTRAINT email_address_pk PRIMARY KEY (oid),
  CONSTRAINT email_address_uniq UNIQUE (oid))
ALTER TABLE email_address ADD CONSTRAINT email_addr_to_person_oid_ref1
  FOREIGN KEY (person_email_addresses) REFERENCES person (oid)
```

3.4 Updating

After the previous change, we can ask UCL-GLORP to update the database schema, causing it to create a table to contain the email addresses. Now, let's suppose that we want to assign two different email addresses to John and we will also take the opportunity to change the street of the address of John:

```
(with-session (*session*)
  (with-unit-of-work ()
    (let ((john (db-read :one 'person)))
      (setf (street (address john)) "33rd Street")
      (setf (email-addresses john)
            (vector
              (make-instance 'email-address
                            :username "012345" :host "freemail.com")
              (make-instance 'email-address
                            :username "john" :host "foo.bar"))))))))
```

⁴ Besides vectors, UCL-GLORP also recognizes type specifiers for lists of elements, including the more relationally-oriented (*one-to-many element-type*) and (*many-to-many element-type*).

This is where a UCL-GLORP's unit of work becomes very useful: instead of forcing us to manually identify the new and changed objects, it automatically computes all changes and writes the proper sequence of updates and inserts to the database. For the previous example, the generated sequence of SQL statements is the following:

```
BEGIN TRANSACTION
SELECT t1.oid, t1.username, t1.host FROM email_address t1
  WHERE (t1.person_email_addresses = 1)
SELECT nextval('email_address_oid_seq') FROM pg_attribute LIMIT 2
UPDATE address SET street = '33rd Street' WHERE oid = 1
INSERT INTO email_address (oid,username,host,person_email_addresses)
  VALUES (1,'012345','freemail.com',1)
INSERT INTO email_address (oid,username,host,person_email_addresses)
  VALUES (2,'john','foo.bar',1)
COMMIT TRANSACTION
```

Note, in the previous SQL code, that a SELECT statement was issued so that UCL-GLORP could compute the changes to the former email addresses of John.

3.5 Querying

One of the best features of UCL-GLORP is the support for combining “normal” Common Lisp code with database queries. As an example, let's suppose we define a predicate that tests that a given person has an email address on a given host:

```
(defun person-with-email-on-host-p (person host)
  (some (lambda (address)
        (string= (host address) host))
        (email-addresses person)))
```

Using this predicate, we can collect all people that have email on, e.g., `freemail.com`:

```
(remove-if-not (lambda (person)
  (person-with-email-on-host-p person "freemail.com"))
  (db-read :all 'person))
```

The previous code reads `:all` people from the database and then filters those that do not satisfy the predicate. To achieve this goal, the `db-read` call starts by generating a generic SQL query that returns all rows from the `person` table and creates the corresponding objects. Then, for *each* person (with `oid` primary key), the `remove-if-not` function calls the predicate that checks the email addresses, causing another SQL query of the form:

```
SELECT t1.oid, t1.username, t1.host
FROM email_address t1
WHERE (t1.person_email_addresses = oid)
```

Clearly, this is a waste of resources because the database might contain hundreds of thousands of rows in the `people` table that will have to be loaded and, for each of them, another query will be issued to compute the corresponding rows from the `email_address` table, thus creating a huge amount of objects just to filter them. Besides the space waste, the process will generate a huge amount of traffic between the application and the database, severely impacting the performance.

Fortunately, a simple rewrite of the expression is sufficient to dramatically speed up the process. To this end, the `db-read` function has a `:where` keyword parameter that accepts the exact same function that the `remove-if-not` accepted. Using this `:where` parameter, the previous expression can be rewritten as:

```
(db-read :all 'person
        :where (lambda (person)
                (person-with-email-on-host-p person "freemail.com")))
```

The results are exactly the same but they are computed differently. Now, the `db-read` call uses the predicate, not to filter the results, but to compute a single SQL query that returns the relevant people in just one database call.⁵

```
SELECT t1.oid, t1.name, t1.address
FROM person t1
WHERE EXISTS (SELECT t2.oid
              FROM email_address t2
              WHERE ((t2.host = 'freemail.com') AND
                    (t1.oid = t2.person_email_addresses)))
```

Given the fact that database communication is considerably slow and that modern database engines have good query optimizers, this second approach will likely run much faster, even taking into account the time needed to analyze the predicate and to translate it into an SQL clause. Not every Common Lisp predicate can be translated into SQL but a representative subset can and Common Lisp programmers will like to know that this subset includes closures. For example, let's suppose that `john` references the "John Adams" that lives in the "33rd Street." Then, the following expression returns all people that live on the same street as `john`:

```
(let ((john ...))
  (db-read :all 'person
          :where (lambda (person)
                  (string= (street (address person))
                          (street (address john))))))
```

⁵ Although the generated SQL uses a subquery, UCL-GLORP can generate joins instead of subqueries just by changing a flag in the configuration of the database connection.

Note, in the `:where` argument of the second `db-read` call, that the function uses the free variable `john`. In this case, the evaluation of the previous expression will make a single database call using the following SQL query:

```
SELECT t1.oid, t1.name, t1.address
FROM (person t1 INNER JOIN address t2 ON (t1.address = t2.oid))
WHERE (t2.street = '33rd Street')
```

Again, this query will run much faster than loading all people and then filter them on the application side. We will discuss the predicate translation process in section 4.

4 From GLORP to UCL-GLORP

During the reincarnation of GLORP as UCL-GLORP, several problems had to be solved in order to overcome the following differences between Smalltalk and Common Lisp:

- In Smalltalk, methods belong to classes and are dispatched according to the class of the receiver. In Common Lisp, methods belong to generic functions and are dispatched according to the type of all the arguments. This is a huge obstacle for the translation because generic functions require congruent methods, while in Smalltalk methods are independent from each other. In practice, each Smalltalk class provides a namespace for its own methods.
- In Smalltalk, the methods of a class have direct access to the instance variables of the receiver. In Common Lisp, this is not possible but can be emulated using the `with-slots` macro. However, a naïve translation from Smalltalk to Common Lisp might end up inserting a `with-slots` form in every method. Replacing `with-slots` with accessors is also not practical because of potential name clashes between the newly created readers and already existent generic functions.
- Smalltalk method invocation protocol makes it easy to explore the proxy design pattern [3]. A proxy class can redefine the default behavior for the `#doesNotUnderstand:` message so that every message sent to the proxy can have a response even when not directly implemented in the proxy class. This is used, for example, to implement the lazy loading of an object: the proxy stands for some not yet loaded object until it receives a message that it doesn't understand, causing it to load the object and forward the message. Common Lisp's generic function invocation protocol makes it much more difficult to implement the same design pattern.
- Smalltalk provides distinct true and false values. On the contrary, Common Lisp amalgamates the false value, the empty list and the symbol `nil` and treats all other values as true.

- Smalltalk provides a distinct null value that is used to initialize instance variables. Common Lisp relies on a different mechanism where instance variables either are unbound or are bound to a value and there is a protocol for accessing those variables (called “slots” in Common Lisp parlance).
- In Smalltalk, collections have identity. Adding or removing elements from collections preserve that identity. Although some Common Lisp collections also preserve identity across modifications, the most used collection data type—the list—was not designed to preserve identity. Usually, this is not a problem to Common Lisp programmers because they tend to respect the Law of Demeter [9], meaning that they don’t directly manipulate containers stored inside some object. However this law is not consistently enforced in Smalltalk programs, where it is not uncommon to see a collection being passed to a method that then modifies it.

It should be clear that there are many more differences but these were the ones that had the biggest impact on the translation of GLORP from Smalltalk to Common Lisp. We will now discuss some of the differences.

5 Slot Access Protocol

UCL-GLORP attempts to be non-intrusive, meaning that it is possible to use plain CLOS classes to define the data model and then map those classes into database tables. One critical point of this mapping is the lazy loading of referenced objects. GLORP implements it using the proxy design pattern. UCL-GLORP implements it using the (non-meta) slot access protocol: each time an object is reconstructed from the information stored in the database, we delay the load of all its associations and the corresponding slots will remain unbound. However, the first time one of those unbound slots is accessed, we detect the unbound slot condition and we identify whether the condition represents a delayed load. In this case, we retrieve the necessary information from the database to construct the delayed object, we store it in the previously unbound slot, and we continue the computation.

This approach requires us to be prepared to handle the unbound slot condition. Obviously, we need to execute all code that potentially needs to access the database in the dynamic scope of an `handler-bind`. This is not problematic because, similarly to the manipulation of files, the managing of database connections already suggests the use of dynamic scope. What is problematic is the reaction to the unbound slot condition because the Common Lisp specification is not sufficiently clear regarding the name (or even existence) of the restart that should be used in that situation. Although one can argue that an `unbound-slot` condition is a subtype of a `cell-error` condition and these errors should have `use-value` and `store-value` restarts, the Hyperspec also includes a short note

mentioning that “No functions defined in this specification are required to provide a `use-value` restart.”⁶

This is an area where we think that the Common Lisp specification should have gone farther and should have specified more conditions and restarts. The hierarchy of conditions presented in the language specification is quite short and makes it difficult to develop portable programs that can handle exceptional situations. The lack of standardized restarts is also an obstacle that could have been more easily removed with a more stringent specification.

It is arguable whether treating exceptional situations as “normal” situations is an adequate approach but, given the fact that Common Lisp is one of the few languages that allow programmatic access to the condition reporting and handling mechanisms, it would be good if those mechanisms were portable across different implementations. It is not a matter of debugging convenience; it is a matter of programming convenience.

6 Function Introspection

Besides mapping object oriented models to relational models, GLORP also maps Smalltalk blocks to SQL statements. Similarly, as was shown in section 3, UCL-GLORP maps functions to SQL statements. To this end, it is necessary to introspect the function so that an abstract syntax tree (AST) can be built in order to rewrite it in terms of database operations.

To construct this AST, GLORP applies the predicate block to an element of a special class that does not implement any of the methods that might be called in the block but that implements the `#doesNotUnderstand:` method so that it records each method that was called, along with its arguments. It then returns another instance of the same special class to continue the construction of the AST. Certain method calls are specially recognized so that other blocks that occur in the code can also be dealt with. Obviously, there is an infinite number of Smalltalk blocks (e.g., all those that cause side-effects) where this introspection strategy cannot possibly work but, in practice, the blocks that need to be introspected are used only as predicates and, usually, these are made of boolean expressions and reader methods that do not cause any side-effects.

Porting this introspection strategy to Common Lisp was exceedingly difficult. Trying to be faithful to the Smalltalk approach, we also used an instance of a special class as predicate argument. However, instead of using the `#doesNotUnderstand:` approach that doesn’t exist in Common Lisp, we used two different approaches. The first one is based on the fact that most generic function calls and, particularly, slot readers, will not be applicable to our spe-

⁶ Independently of what the specification says, at least one important Common Lisp implementation didn’t provide the correct restarts for the `unbound-slot` condition.

cial instance.⁷ When the error is detected, we immediately define an additional method that specializes the generic function in question for our special class (so that it registers the call) and we invoke the `continue` restart so that the call is indeed registered and the introspection process can proceed.

Unfortunately, this contorted scheme cannot work with non-generic functions because it critically depends on the `continue` restart that is not generally available and, moreover, it can't detect the use of boolean operators because (1) `and` and `or` are macros that expand into special forms and (2) `not` accepts anything as argument, never signaling any error. This is where our second approach is applied: we shadow those symbols and provide different implementations so that we can have an handle on their evaluation and we also do this for all non-generic functions that might occur in a predicate that will be used for restricting a database query, such as the `some` and `string=` functions that we presented in section 3. Although it is not measurable in our experiments, we are aware that replacing (normal) functions with their generic counterparts might have a considerable impact on the performance. However, without these drastic measures, we found it highly difficult to introspect Common Lisp functions.

7 Conclusions and Related Work

In this paper, we presented UCL-GLORP, a Common Lisp reimplementation of GLORP, an well-established ORM for Smalltalk. UCL-GLORP differs from GLORP in several important ways:

- UCL-GLORP infers models and mappings from a set of CLOS classes. This is something that is beyond GLORP capabilities because, contrary to CLOS, Smalltalk classes do not have any standardized way of annotating slots with the necessary type information.
- UCL-GLORP never expose proxies. Instead, these are completely hidden from application code and are resolved whenever we trap the `unbound-slot` condition associated with the corresponding slot access. This is a much safer approach to lazy loading because, contrary to GLORP, it is impossible, in UCL-GLORP, to create identity problems between a proxy and the object it stands for.
- UCL-GLORP is more complex than GLORP because we need to deal with a lot more diversity in Common Lisp than in Smalltalk. One of the strongest points of Smalltalk is, indeed, its simplicity and uniformity that makes it easier to centralize behavior.

⁷ Unfortunately, the ANSI Common Lisp specification does not specify the subtype of error that should be signaled and all the implementations tested simply signal an instance of `error` with different error messages.

- There is no support in GLORP for schema evolution. UCL-GLORP, on the contrary, provides such support. Besides mapping a set of classes into a set of database tables, UCL-GLORP is also capable of mapping a set of class *changes* into a set of database *changes*. Sometimes, there is more than one way to do this and whenever this happens, UCL-GLORP presents the different options and requests guidance from the programmer.

One of the main responsibilities of an ORM is to ensure the persistence of data. In the Common Lisp camp, this task can also be accomplished using any of the other persistence frameworks, namely, UCL+P [4], Stalice [12], PCLOS [10], PLOB! [6], AllegroCache [1] and many others. However, besides ensuring persistence, an ORM also ensures that persistent data is stored according to the principles of the relational model. This is a much more complex requirement and, at the time we start developing UCL-GLORP, there was no ORM for Common Lisp that would allow us to non-intrusively provide a mapping between CLOS classes and relational databases.

Very recently, another ORM for Common Lisp was presented: CL-PEREC [8]. Although it targets the same goals, CL-PEREC and UCL-GLORP have several important differences:

- In CL-PEREC, classes whose instances should be persistent must belong to a special metaclass. There is no such requirement in UCL-GLORP and plain CLOS instances can be made persistent without any changes.
- CL-PEREC does not include relations in the class definitions. Instead, all relations must be defined separately. UCL-GLORP, on the other hand, can infer the relations from the class definitions.
- CL-PEREC provides a specialized SQL-like query language. Although we didn't mention it in this article, UCL-GLORP also provides an SQL-like language but this language is not generally used by the programmer. Instead, it is used as the target for the translation of Common Lisp functions that restrict the queries.
- Contrary to UCL-GLORP, CL-PEREC doesn't implement units of work and transactions are not supported on the object level. This means that every slot update is immediately transferred to the database, thus preventing the optimizations and reorderings that are done by UCL-GLORP.

Besides the mentioned differences, there is a more profound mismatch between CL-PEREC and UCL-GLORP: CL-PEREC provides a new language for class definitions and queries while UCL-GLORP tries very hard to remain faithful to the “normal” CLOS style. We think we achieved this goal because, using

UCL-GLORP, programs using CLOS can be made persistent without requiring incompatible changes. This is an important property because it makes the program independent of the persistency backend used.

Being non-intrusive is a fundamental goal for UCL-GLORP but it might make it more difficult to implement optimizations that take advantage of certain usage patterns. If these optimizations are critical, the solution is to manually provide the models and mappings and to use the low-level UCL-GLORP SQL interface, using a more persistency-aware development model.

Although there are still several rough edges that we would like to smooth, UCL-GLORP is perfectly usable and, in fact, we have been using UCL-GLORP in a production environment for more than a year, to provide the persistency layer of a web-based application. On the negative side, it should be mentioned that portability is UCL-GLORP major problem because it stresses Common Lisp in ways that are not well-defined in the language specification. At the moment, UCL-GLORP only runs in Allegro Common Lisp and Lispworks.

References

1. Jans Aasman. AllegroCache: A high-performance object database for large complex problems. In *5th International Lisp Conference*, Stanford University, June 2005.
2. Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2005.
3. Gamma, Helm, Johnson, and Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 2000.
4. J. H. Jacobs and Mark R. Swanson. UCL+P - defining and implementing persistent common lisp. *Lisp and Symbolic Computation*, 10(1):5–38, 1997.
5. Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
6. Heiko Kirschke. Persistency in a dynamic object-oriented programming language. Technical Report 10, University of Hamburg Computer Science Department, Jul 1995.
7. Alan Knight. Glorp: generic lightweight object-relational persistence. In *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 173–174, New York, NY, USA, 2000. ACM.
8. Attila Lendvai, Levente Mészáros, and Tamás Borbély. cl-perec: RDBMS based CLOS persistency. <http://common-lisp.net/project/cl-perec/>, Feb 2008.
9. K. J. Lienberherr. Formulations and benefits of the law of demeter. *SIGPLAN Not.*, 24(3):67–78, 1989.
10. Andreas Paepcke. PCLOS: A Flexible Implementation of CLOS Persistence. In S. Gjessing and K. Nygaard, editors, *Proceedings of the European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science, Springer Verlag, 1988.
11. Kevin M. Rosenberg. CLSQL – a multi-platform SQL interface for Common Lisp. <http://clsql.b9.com/>, September 2007.
12. D. Weinreb, N. Feinberg, D. Gerson, and C. Lamb. An object-oriented database system to support an integrated programming environment. *Data Engineering*, 11(2):33–43, June 1988.

An Implementation of CLIM Presentation Types

Timothy Moore
Red Hat SARL
timoore@redhat.com

Abstract: Presentation types are used in the CLIM interface library to tag graphical output with a type and establish an input type context in which the user may use the keyboard to type input, accepted by a parser associated with that presentation type, or click on the graphical representation of an object that has an appropriate presentation type. Presentation types are defined using a syntax reminiscent of the `deftype` syntax of Common Lisp; the input and output actions of the types, as well as aspects of their inheritance, are implemented using a system of generic functions and methods directly based on CLOS. The presentation type system is different enough from the Common Lisp type system that its types, generic functions and methods do not map directly to those of Common Lisp. We describe presentation types implementation in McCLIM which uses the CLOS Metaobject Protocol to implement presentation type inheritance, method dispatch and method combination without implementing an entire parallel object system next to CLOS. Our implementation supports all types of method combination in the presentation methods, including user-defined method combination.

Key Words: Common Lisp, CLIM, presentation types, metaobject protocol

1 Introduction

The specification of the Common Lisp Interface Manager (CLIM) (McK; RYD91) describes a graphical interface toolkit for Common Lisp (AI96) in which program objects are explicitly associated with graphical representations of those objects, called *presentations*. Based on the user's interactive input, and according to a context of desired input established by the program, the objects are made available as input to commands, either implicitly in the traditional style of GUI interaction or explicitly via a command line. Presentations store a *presentation type* as well as an object, and it is this type that is used to test whether a presentation can satisfy the current input context. In most programs that use CLIM a highlight is drawn around objects that match the current input context as the user moves the mouse over them, and a message summarizing the input action that will occur if the mouse buttons are pressed is displayed at the bottom of the screen. Presentation types share similarities both with Common Lisp built-in types and with Common Lisp Object System (CLOS) classes. A system of generic functions and methods, also similar to that in Common Lisp, supports dispatch on types as if they were objects. Several of these *presentation generic functions* are defined by CLIM to control input parsing and output of the objects associated with presentation types, type membership tests, and subtype relations of the presentation types.

The semantics of presentation types are different enough from those of Common Lisp types defined via the `deftype` macro and standard classes that a naïve implementation would duplicate a lot of the complex method combination and dispatch code that must exist in a Common Lisp implementation to support CLOS. We describe here the the implementation of the presentation type system in McCLIM(SM02), an open source implementation of CLIM. We used the Metaobject Protocol(KdR91) present in most Common Lisp implementations to implement presentation types and generic functions. McCLIM was written from scratch with reference to the CLIM specification, a terse and, at times, incomplete and contradictory document, and a few available example CLIM programs. For an introduction to Common Lisp and CLOS refer to texts such as (Sei05; Gra99; Nor91; KG89). An introduction to CLIM can be found in (RYD91; Møl).

2 Presentation Types

Before describing the definition of presentation types, it is useful to review the ways that new types, called *type specifiers*, are defined in Common Lisp because CLIM presentation types use concepts from these approaches. A type specifier is a name or a list of a name and parameters that can be passed to `typep` to test whether an object is of a certain type or to `subtypep` to determine subtype relationships. Type specifiers are defined using either the `deftype` macro or the `defclass` macro. Instances of the classes defined using `defclass` can be created using `make-instance` and the arguments specified in the defining `defclass` form. The type specifier of a user-defined class is either the name of the class (as a symbol) or a *metaclass* object created by the system to represent the type.

We ignore types created with the Common Lisp macros `defstruct` and `define-condition` as they are very similar to classes defined with `defclass`.

2.1 Common Lisp type specifiers and `deftype`

The `deftype` macro defines a function that expands a type specifier into another type specifier through a process very similar to macro expansion; indeed, the function created by `deftype` behaves exactly like a macro expander function created by `defmacro`, except that the default argument for optional and keyword arguments in the type specifier form is `*`, the wildcard type specifier. The body of the `deftype` form returns a new type specifier using the arguments, existing type specifiers, compound type specifiers like `and` and `or` that create intersections and unions of existing types, or the `satisfies` type specifier that uses a function predicate to define a type. Figure 1 shows a simple `deftype` definition that creates a subset of the `integer` type with parameters to `integer` and a functional predicate. It is important to note that the types created with `deftype` cannot specify objects with new characteristics in Common Lisp; they can only restrict

```
(deftype even-positive-integer (&optional high)
  '(and (integer 0 ,high) (satisfies evenp)))
```

Figure 1: Example of `deftype` usage

```
(defclass person ()
  ((name :accessor name :initarg :name)
   (age :accessor age :initarg :age)))
```

Figure 2: class definition example

existing types by giving them explicit parameters or perform set operations on the membership of the types. They cannot be specified in `defmethod` argument specializers.

2.2 Classes defined with `defclass`

Classes are user-defined types that have superclasses and that can store data in *slots*. A class is defined using the `defclass` macro. A *slot definition* specifies the name of the slot and optional parameters such as the type of the value of the slot and the names of generic functions that get and set its value.

Figure 2 shows the definition of a simple class. This class is named `person` and has two slots, `name` and `age`. Classes can inherit from one or more user-defined classes to create a subtype relationship. The new subclass is a subtype of its superclasses. Figure 3 shows the definition of an `engineer` class that inherits from a `specialty-mixin` class as well as from the `person` class.

```
(defclass specialty-mixin ()
  ((specialty :accessor specialty :initarg :specialty)))

(defclass engineer (person specialty-mixin)
  ())

(defclass cook (person specialty-mixin)
  ())
```

Figure 3: multiple inheritance with a mixin class

```

(define-presentation-type integer (&optional low high)
  :options ((base 10) radix)
  :inherit-from '((rational ,low ,high) :base ,base :radix ,radix))

(define-presentation-method presentation-typep (object (type integer))
  (and (integerp object)
       (or (eq low '*)
           (<= low object))
       (or (eq high '*)
           (<= object high))))

(defmethod presentation-type-of ((object integer))
  'integer)

(presentation-typep 42 '(integer 6 43))
T

```

Figure 4: Example of presentation type, its definition, and a presentation method

2.3 define-presentation-type

Presentation types combine aspects of type specifiers and classes considered as types. The type is descriptive and parameterized, like a type specifier, but is not instantiable. The concrete representation of a presentation type is either a symbol or a list with arguments. Presentation types support multiple inheritance, and can participate in a kind of method dispatch and combination in which parameters of the type are available inside the methods.

Figure 4 shows the definition of a presentation type, `integer`, that is a part of CLIM. The parameters `low` and `high` specify the members of the type. This type also specifies options that do not affect type tests and membership but do affect how presentations with this type will be displayed and how input will be parsed in this input context. The `:inherit-from` argument is a form that specifies the supertypes of the presentation type and can use the parameters and options as arguments in a limited way: the form must be able to create its result without referring to the actual value of the parameters and options. This allows the `:inherit-from` form to be analysed using dummy arguments at the time of the type definition.

To support dispatching on a single presentation type argument, CLIM provides *presentation generic functions* and *presentation methods* that are similar to their CLOS equivalents – for example, method combination and effective method

computation work as expected – but that also make parameters and options available as implicitly defined variables in the methods, properly transformed for the presentation type. This style of magic slot access in methods is not found elsewhere in Common Lisp today but is retained for compatibility with an earlier presentation-based system found in Dynamic Windows in the Symbolics Genera environment.¹ In Figure 4, `presentation-typep` is a presentation generic function defined by CLIM. The `type` argument is a presentation type. This method is properly sorted with respect to other applicable presentation methods such as, for example, a method for the presentation type `rational`. The parameters and options of the presentation type are available as bound variables inside the method.

The call to `presentation-typep` in Figure 4 shows a typical use of presentation types. `presentation-typep` is a function that invokes the presentation generic function `presentation-typep`. This computes and invokes the effective method for this call which then calls the presentation method for the type `integer`. Figure 5 shows two more basic presentation methods, `present` for output and `accept` for input, that make use of the options available in presentation types.

CLOS class names and metaclass objects are valid as presentation types. Many builtin Lisp types have a presentation type equivalent with the same name.

In order to make presentation types less abstract, Figure 6 shows some experiments with presentation types in the Listener application supplied with McCLIM. The `present` function writes output annotated with a presentation type, called a *presentation*, to an output stream. The function `accept` reads input, either typed by the user or entered by clicking on a presentation with a compatible presentation type. In this case “42” is acceptable because the presentation type (`integer 0 50`) is a subtype of (`real 0 100`). The pointer documentation pane at the bottom of the listener window shows the action if the user clicks on the left mouse button: “42” will be accepted.

3 Implementation of Presentation Types

3.1 Presentation Types and Presentation Methods

Although they are represented as lists, presentation types have many characteristics of CLOS objects. Their parameters and options are similar to class slots, and they have an inheritance relation with their supertypes. However, parameters and options are not inherited from supertypes – they parameterize the supertypes and they may be arbitrarily transformed within the limits

¹ The designer of this feature now says “I can now say that this was a mistake, and that we should have simply implemented a `with-slots`-like macro that did the right thing.”(McK08)

```

(define-presentation-method present (object (type integer) stream
      (view textual-view)
      &key acceptably for-context-type)
  (declare (ignore acceptably for-context-type))
  (let ((*print-base* base)
        (*print-radix* radix))
    (princ object stream)))

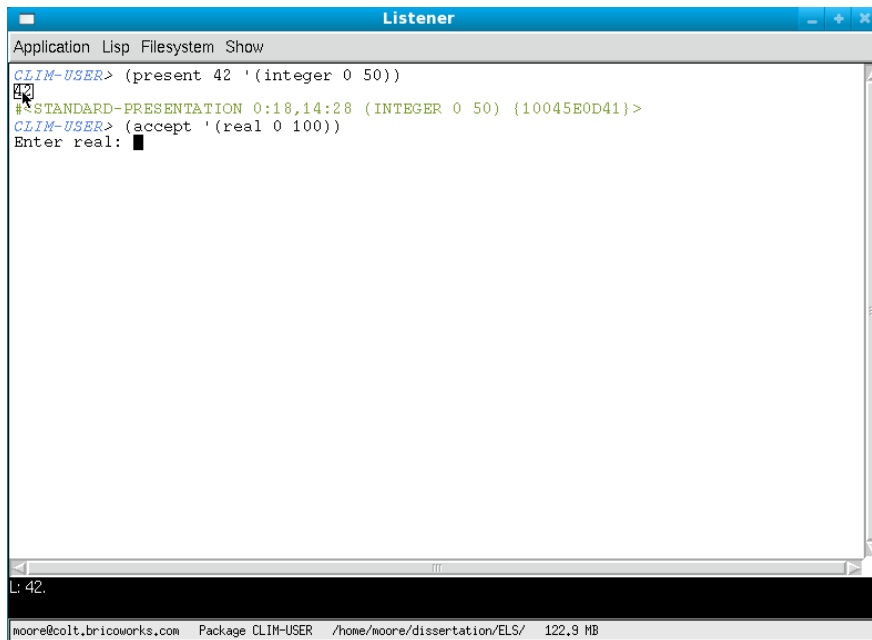
(define-presentation-method accept ((type integer)
      stream (view textual-view)
      &key (default nil defaultp)
      default-type)
  (let ((*read-base* base))
    (let* ((token (read-token stream)))
      (when (and (zerop (length token))
                  defaultp)
        (return-from accept (values default default-type)))
      (parse-integer token))))

```

Figure 5: Example `present` and `accept` methods. The presentation options `base` and `radix` are used in these methods.

imposed on the `:inherit-from` specification. A parameter may have a different value in a presentation method written on a supertype than it does in a subtype method; this is the opposite of the behavior of slots, which have a single value in an object. Nevertheless, if a presentation type could be represented as a CLOS object, then presentation method dispatch could be implemented easily using normal CLOS method dispatch. The CLIM specification seems to point in this direction, saying “Every presentation type is associated with a CLOS class... `define-presentation-type` defines a class with meta-class `presentation-type-class` and superclasses determined by the presentation type definition.” Also, the lambda list of a presentation generic function must contain a mysterious “`type-key` or `type-class` [argument]; this argument is used by CLIM to implement method dispatching.”

There are some awkward complications with this approach. It is easy to construct a type key for presentation types defined via `define-presentation-type`; it can be created as part of the evaluation of the defining form. But CLOS classes are implicitly presentation types too, and it is not obvious how to create an instance of an arbitrary class without any knowledge of its arguments. It is reasonable to define presentation methods on `standard-object`, the superclass

A screenshot of a Lisp Listener window. The window title is "Listener". The menu bar includes "Application", "Lisp", "Filesystem", and "Show". The main text area shows the following interaction:

```
CLIM-USER> (present 42 '(integer 0 50))
42
#STANDARD-PRESENTATION 0:18,14:28 (INTEGER 0 50) {10045E0D41}>
CLIM-USER> (accept '(real 0 100))
Enter real: █
```

The status bar at the bottom shows the path "/home/moore/dissertation/ELS/" and the package "CLIM-USER".

Figure 6: Presentation types in action. “42” has been presented to the screen with a presentation type that is a subtype of `integer`; that value can be accepted if a subtype of `real` is requested.

of all CLOS classes, but many presentation types do not have `standard-object` as a supertype and so those methods should not be applicable when a presentation generic function is called on such a type.

3.2 The Metaobject Protocol

Fortunately most implementations of Common Lisp implement the Metaobject Protocol, or MOP, as described in (KdR91). This exposes many of the internal details of class definition, generic function definition and method dispatch and allows them to be customized. The implementation of presentation types makes use of two major features of the MOP. The MOP specifies that a *class prototype* object, which is an instance of a class with undefined slot values, exists for all classes. This is obviously ideal to use as the type key object. Also, the MOP supports broad customization of the selection of applicable methods in a generic function call via the generic functions `compute-applicable-methods` and `compute-applicable-methods-using-classes`. Even Common Lisp implementations that do not support the full MOP usually have some internal

functionality that is equivalent to these features and that can be used in the presentation types implementation.²

3.3 Implementation

A class metaobject of type `presentation-type-class`, a subclass of `standard-class`, is created for each defined presentation type. The class is given a fake name so that there is no conflict between presentation types and built-in types of the same name. This class stores details about the presentation type including a function that produces the `:inherit-from` form from parameter and option arguments. The supertypes of the presentation type, retrieved by running the `:inherit-from` function with dummy arguments, become the direct superclasses of the metaobject. A hash table maps presentation type names to these metaobjects. CLOS classes that are mentioned in `define-presentation-type` forms are represented by a presentation type class that is not a metaclass but that does contain a reference to the metaclass of that class.

According to the CLIM specification(McK), presentation generic functions are called using the macros `funcall-presentation-generic-function` and `apply-presentation-generic-function`. This extra syntax is rather awkward but, in actual CLIM programming, presentation generic functions are not called directly by the programmer; they are invoked indirectly by calling functions defined in the CLIM specification. For example, a program calls the `present` function, and that function calls the presentation generic function of the same name, perhaps after establishing dynamic state and defaulting arguments. One of the arguments in the presentation generic function call will be a presentation type specifier which is examined to find the presentation type metaobject and thence the associated class prototype. This is passed as an argument to the presentation generic function as the type key object. If the presentation type argument is a CLOS class, that class' prototype is passed.

The type key object, and all the other arguments of the presentation generic function, are used to compute the applicable methods for the function invocation. Two generic functions in the Metaobject Protocol, `compute-applicable-methods` and `compute-applicable-methods-using-classes`, are specified as potentially being called when figuring the applicable methods for a particular function invocation, so any customization of this process must define methods for both. The presentation generic function class, a subclass of `standard-generic-function`, has specialized versions of these two methods which eliminate any potentially applicable method that is specialized on `standard-object` if the presentation type argument is not a CLOS class, as shown in Figure 7. The

² This work was originally done using OpenMCL, which at the time did not have a full MOP implementation.

```

(defmethod compute-applicable-methods :around
  ((gf presentation-generic-function) arguments)
  (let ((methods (call-next-method)))
    (if (typep (class-of (car arguments)) 'presentation-type-class)
        (remove-if #'(lambda (method)
                       (eq (car (clim-mop:method-specializers method))
                           *standard-object-class*))
                    methods)
        methods)))
  methods)))

```

Figure 7: `compute-applicable-methods` implementation which removes any presentation methods defined for CLOS types from methods for a non-CLOS presentation type. The code for `compute-applicable-methods-using-classes` is similar.

presentation methods themselves are just regular methods with an additional argument for the type key.

The body of a presentation method is wrapped by code that expands the presentation type argument from its actual type to the supertype expected by the method. Each presentation type's `:inherit-from` function can translate a type specifier to that of its supertypes; this can be done repeatedly on a subtype and its supers until the type specifier of an arbitrary supertype is produced. Once this is in hand, the parameter and options are decoded and bound to variables in the method body. Figure 8 shows the expansion of the method definition in Figure 4.

In effective methods that contain many constituent methods this strategy could lead to poor performance because the expansion functions for the most specific classes need to be run repeatedly as less specific methods are called. It was thought that it would be useful to introduce a caching mechanism to mitigate this effect, but profiling has not shown this process to be a bottleneck in real applications that use McCLIM. An alternate strategy would be to perform the expansion outside of the method body, in the method combination code. This approach avoids unnecessary expansion, but it breaks all non-standard method combination. The simpler approach used in McCLIM, which keeps the type argument expansion inside the method body, allows all standard and user-defined method combination to “just work.”

4 Conclusion

We have explored the implementation of a complex part of the CLIM specification, presentation types, using features of the Common Lisp Metaobject

```

(defmethod %presentation-typep
  ((type-key |(presentation-type common-lisp::integer)|)
   object type)
  (block presentation-typep
    (let ((#:massaged-type2397
          (translate-specifier-for-type
            (type-name-from-type-key type-key) 'integer type)))
      (let ((parameters (decode-parameters #:massaged-type2397)))
        (declare (ignorable parameters))
        (with-presentation-type-parameters
          (integer #:massaged-type2397)
          (and (integerp object) (or (eq low '*) (<= low object))
              (or (eq high '*) (<= object high))))))))))

```

Figure 8: The expansion of `define-presentation-method`.

Protocol. The implementation of presentation method dispatch, which uses class prototypes, turns out to be reasonably straightforward; the major remaining complexity is in the translation of presentation subtype parameters to parameters for the supertypes. At the time that CLIM was first implemented and specified (1992), the MOP was quite new and not well supported in CLOS implementations. If the MOP had been well supported, some syntactic choices in CLIM (such as the `funcall-presentation-generic-function` macro) would have undoubtedly been different, and the specification could have referenced MOP concepts such as the class prototype directly (McK08). Without MOP support the implementation of presentation types and presentation method dispatch would require an enormous amount of coding; in fact this daunting task had blocked progress in the McCLIM project. The realization that a small part of the Metaobject Protocol can be used to implement this part of CLIM resulted in a robust presentation type system for McCLIM in a fairly short time. This in turn supports a large amount of functionality, including presentation methods for standard types and the full machinery for `accept` and `present`, that make McCLIM a real implementation of CLIM.

4.1 Acknowledgements

I would like to thank Robert Strandh, who invited me to be a “professeur associé” at the Université de Bordeaux. As a result of this life-changing event much work was done on McCLIM. Also, I would like to thank the entire McCLIM team, in particular Christophe Rhodes and Troels Henriksen, for their bug fixes

to the presentation type code in McCLIM. McCLIM can be found at <http://common-lisp.net/project/mcclim/>.

References

- American National Standards Institute and Information Technology Industry Council. *American National Standard for Information Technology: programming language — Common LISP*. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1996. Approved December 8, 1994.
- Paul Graham. *ANSI Common LISP*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition, 1999.
- Gregor Kiczales and Jim des Rivieres. *The art of the metaobject protocol*. MIT Press, Cambridge, MA, USA, 1991.
- Sonya E. Keene and Dan Gerson. *Object-oriented programming in Common LISP: a programmer's guide to CLOS*. Addison-Wesley, Reading, MA, USA, 1989.
- Scott McKay. *Common Lisp Interface Manager CLIM II Specification*. Available at <http://www.stud.uni-karlsruhe.de/~unk6/clim-spec/>.
- Scott McKay. personal communication, April 2008.
- Ralf Möller. User interface management systems: the CLIM perspective. <http://www.sts.tu-harburg.de/~r.f.moeller/uims-clim/clim-intro.html>.
- Peter Norvig. *Paradigms of artificial intelligence programming: case studies in Common LISP*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1991.
- Ramana Rao, William M. York, and Dennis Doughty. A guided tour of the Common Lisp interface manager. *SIGPLAN Lisp Pointers*, IV(1), 1991. Updated 2006 by Clemens Frühwirth.
- Peter Seibel. *Practical Common Lisp*. Apress, 2005.
- Robert Strandh and Timothy Moore. A free implementation of CLIM. 2002.

Custom Specializers in Object-Oriented Lisp

Jim Newton
Cadence Design Systems
Mozartstrasse 2
D-85622 Feldkirchen Germany
jimka@cadence.com

Christophe Rhodes
Department of Computing
Goldsmiths, University of London
New Cross, London, SE14 6NW
c.rhodes@gold.ac.uk

Abstract: We describe in this paper the implementation and use of custom specializers in two current dialects of Lisp: SKILL and Common Lisp. We motivate the need for such specializers by appealing to clarity of expression, referring to experience in existing industrial applications. We discuss the implementation details of such user-defined specializers in both dialects of Lisp, detailing open problems with those implementations, and we sketch ideas for solving them.

1 Introduction

Lisp has a venerable history of object-oriented programming; at one point in time, early in the history of object-orientation, Flavors [Moo86] and New Flavors, Common Objects, Object Lisp and Common Loops [BKK⁺86] all coexisted. The Common Lisp Object System (CLOS) was incorporated into the language in June 1988 [Ste90, Chapter 26], and when the ANSI Common Lisp standard [PC94] was formalized in 1995, Common Lisp became the first ANSI-standardized programming language with support for object-oriented programming.

In the object systems in the Lisps under discussion in this paper, method specializers have the function of determining whether a particular method is applicable to a set of function arguments or not; method qualifiers determine the function of the method within the effective method (from method combination) if the method is applicable at all.

In standard Lisps, the repertoire of specializers is limited: in SKILL, only classes are allowed as specializers by default, matching instances of that class; in Common Lisp, classes and `eq1` specializers (matching a single object by identity) are allowed by default, though the CLOS Metaobject Protocol (MOP) allows for extensibility in principle, as it specifies a `mop:specializer` metaobject class.

1.1 Custom Specializers

It is sometimes the case that applications require dispatch on objects whose behaviour is not separated by class structure; the dispatch may be influenced by the global application state, or by the values of slots in the objects, or other such factors. In object systems where the specializer metaobject class is not extensible, there is then an impedance mismatch between the expression of the functionality and its implementation, and it is this impedance mismatch that we address by allowing the user to define subclasses of the specializer class. By giving the user this option, we aim to provide a means to improve locality and clarity of the implementation of a particular solution to a problem, by allowing direct expression rather than manual reimplementing of dispatch machinery to distinguish between things that happen to be instances of the same Lisp class (or where the class of the object is not relevant for dispatch).

This paper discusses the use and implementation of metaobject protocols to allow the user to take advantage of the ability to define subclasses of the specializer class; after introducing some background and discussing related work in the next section, we present a worked example in section 3 to attempt to motivate the definition and use of such specializer metaobject classes. We discuss implementation issues regarding both SKILL and Common Lisp in section 4, and conclude in section 5.

2 Background

2.1 The SKILL Programming Language

The users of Cadence Design Systems' custom Integrated Circuit (IC) tools use the SKILL® programming language [Bar90, Pet93] extensively. Programmers write applications which customize the look and feel of the graphical system, automate the design process by reducing the amount of repetitive work the design engineer must do, and perform time-consuming, tedious verification checks. Other types of programs include automatic layout generation which quickly produce parameterizable layouts which are correct by design. The language has an optional C-style syntax with many engineer-friendly shortcuts, making it easy for non-programmers to write simple scripts to help in their daily work.

The same language is also a Lisp system having the basic features one would expect: a Read-Eval-Print Loop (REPL), a debugger, garbage collection, lexical and dynamic scoping, macros, and anonymous functions. As with most Lisp systems, the language can be extended through adding functions to the runtime environment.

The SKILL language has a built-in object system called the SKILL++ Object System or simply SKILL++. SKILL++ is based on CLOS, but provides only a

subset of the capabilities; missing are features such as: multiple dispatch, multiple inheritance, method combination, method qualifiers, equivalence specializers, and a Metaobject Protocol. Instead, it provides single dispatch, single inheritance, analogues to Common Lisp's `call-next-method` and `next-method-p`, class and method redefinition, explicit environment objects, and a per-method choice between lexical and dynamic scoping. Also important to note is that while the language is interpreted by a proprietary virtual machine, the method dispatch mechanism in particular is implemented in a high performance compiled language; consequently, generic function calls are as fast as normal function calls.

It should be stressed that, although SKILL is a special-purpose language environment and exists primarily within proprietary applications, it has a wide user base, as a substantial fraction of the world's IC design software is provided by Cadence Design Systems; many of the chips in today's consumer devices have been simulated or designed within a SKILL-based system. Thus, there is considerable potential benefit from learning from language design experience, both to improve SKILL itself and to make language innovations developed for SKILL environments available to Common Lisp users.

2.2 Common Lisp

CLOS was developed in conjunction with the design of a Metaobject Protocol (MOP), described in *The Art of the Metaobject Protocol* (AMOP) [KdRB91]. Common Lisp as standardized only includes a very small portion of this Metaobject Protocol (for instance, a recommendation to use `mop:slot-value-using-class` in `slot-value`; some introspective functionality such as `find-method`; and arguably a little ability for intercession in `compute-applicable-methods`, though in fact the standard does not require that `compute-applicable-methods` be called as part of generic function dispatch), and so to customize the behaviour of the object system in Common Lisp it is necessary to go beyond the standard language.

Many Common Lisp implementations support some of the MOP, to varying extents; a survey from a few years ago [BdL00] revealed many aspects of MOP support as being incomplete, even at the coarse level of specified classes and generic functions being unimplemented. More recently, the Closer¹ project has provided both a set of test cases for implementations of the Metaobject Protocol – which has encouraged some implementations to enhance their support for it² – and a compatibility layer to provide an environment as close as possible to that described in AMOP in major implementations of Common Lisp.

¹ <http://common-lisp.net/project/closer/>

² At the time of writing, the MOP implementation of Steel Bank Common Lisp [N⁺00] fails none of tests in the Closer MOP suite.

```
(defgeneric walk (expr env call-stack)
  (:generic-function-class sop-cons-generic-function))
```

Figure 1: Code walker generic function definition.

2.3 Related Work

The issue of dispatch customization in Common Lisp has arisen before; for example, predicate dispatching in Common Lisp has been discussed in [Uck01]. In that work, the predicate was not restricted at all, and the solution presented involved extending method qualifiers (arbitrary predicates not being associated with any particular argument, and methods being distinguished from each other only on the basis of qualifiers and specializers). Portability difficulties with this approach were noted at the time, and would likely still be present today; for example, some implementations will only accept non-standard qualifiers if the generic function has a non-standard method combination. Strictly, `define-method-combination` will signal errors if methods are placed in the same method group having the same specializers (even if the intent is to use qualifiers to influence method applicability): qualifiers in Common Lisp are meant to affect method combination rather than method selection.

Predicate dispatch in other languages has also been investigated; a system has been presented and implemented for Java [Mil04], wherein the predicates affecting dispatch are restricted to a set which can be reasoned over, and for which ambiguities are forbidden in the selection of the most specific method. We prefer to leave such policy decisions to the users of the system, at least while the capabilities and expressiveness are being explored: if it turns out that restricting specializers to express a limited set of predicates is acceptable, that can be enforced at a later stage.

At this time, we make no attempt to implement a specific predicate dispatch mechanism in either SKILL or Common Lisp, but rather aim to provide a framework which is both sufficiently general to express predicate dispatch and straightforward to use, allowing issues of determinism, portability and performance to be explored and addressed by users.

3 Using Custom Specializers: a Worked Example

The following excerpts are from a code walker expressed using custom specializers. The code walker examines code written in a particular Lisp dialect and reports unbound and unused variables. For purposes of simplicity, the illustrated

```

(defmethod walk ((expr list) env call-stack)
  (let ((call-stack (cons expr call-stack)))
    (walk (car expr) env call-stack)
    (walk (cdr expr) env call-stack)))

(defmethod walk ((expr (eql nil)) env call-stack)
  nil)

(defmethod walk ((expr t) env call-stack)
  (format t "invalid expression ~A: ~A: ~A~%"
    (class-name (class-of expr)) expr call-stack))

```

Figure 2: Recursion engine and termination condition

implementation uses a Common Lisp-like syntax, with SKILL-like semantics in one or two respects noted below.

The goal of this illustration is to give an example of a solution that is more parsimonious when the language supports describing actions on wider ranges of data, rather than to convince that a particular type of specializer (such as the `cons` specializer used here) itself is a good idea. As with any pedagogical example, the same application could be written in many different ways without great loss of clarity.

The form in figure 1 defines the generic function `walk` as an instance of the generic function class named `sop-cons-generic-function`, which is assumed to already exist. We discuss the implementation issues of this metaclass in section 4.1.

The implementation of `walk` we present here contains four conceptual parts:

- a recursion engine which includes a termination condition and error handling;
- code to recognize variable references and mark bindings as *used*;
- code to ignore all irrelevant forms encountered during the recursion;
- code to handle special forms.

We begin by implementing the first three parts using standard CLOS functionality; the part to handle special forms is then implemented using a non-standard subclass of `mop:specializer`.

3.1 Code Walker Framework

The main engine of the code walker (figure 2) starts at a top level expression. If the expression is a list, it calls itself recursively on the elements of the list – with

```

(defmethod walk ((var symbol) env call-stack)
  (if-let (binding (find-binding env var))
    (setf (used binding) t)
    (format t "unbound: ~A: ~A~%" var call-stack)))

```

Figure 3: Checking the bindings of symbols.

a few notable exceptions. Some of the necessary exceptions can be handled by equivalence specializers such as (`eq1 t`) and (`eq1 nil`). Lisp special forms, such as (`quote ...`) and (`lambda ...`) forms, cannot be described by equivalence specializers but can be with `cons` specializers.

Next is the traversal engine based on the class specializer `list` and the termination condition based on an equivalence specializer (`eq1 nil`). Thus the engine keeps traversing the lists until they are exhausted. There is also a method specializing on class `t` which will be called if something is encountered which the code walker cannot otherwise handle. The job of the methods that follow will be to assure that everything that occurs in the traversal is handled by an appropriate method and that the "invalid expression" message never gets printed.

When a symbol is encountered the method in figure 3 is applicable. A check is made to see whether the variable is bound in the environment³. If so, the `used` slot of the binding object is set to true, to note that the binding is used. If the variable is unbound, then a diagnostic message is emitted, informing the user of where the reference to an unbound variable is made.

Figure 4 shows how certain types of self-evaluating atoms such as strings, numbers, and the symbol `t` are simply ignored when searching for variable references. A full implementation of this would ignore all atoms which cannot name variables; in this restricted Common Lisp-like language, we assume that those objects are instances of either `string` or `number`.

3.2 Special Forms

We now implement some of the special forms. Note that `quote` and `lambda` themselves are not special forms; they are simply symbols which evaluate as any other symbol – if one of these symbols is encountered in a context where it is used as a variable, the code walker must treat it as such. This means we cannot write a method for `walk` specializing on (`eq1 quote`)⁴. However, lists for evaluation

³ The implementation of the `find-binding` function is omitted. It returns a *binding* object by searching for a named variable in a given *environment* object. Such a binding object has an accessor named `used` to hold a boolean, indicating whether the binding is used or not.

⁴ Note that unlike in Common Lisp, here the argument of the `eq1` specializer is unevaluated; (`eq1 quote`) is correct, rather than (`eq1 'quote`). We discuss this further in

```

(defmethod walk ((expr string) env call-stack)
  nil)
(defmethod walk ((expr number) env call-stack)
  nil)
(defmethod walk ((expr (eql t)) env call-stack)
  nil)

```

Figure 4: Ignoring certain atoms.

```

(defmethod walk ((form (cons (eql quote))) env call-stack)
  nil)

(defmethod walk ((form (cons (eql lambda))) env call-stack)
  (destructuring-bind (lambda lambda-list &rest body) form
    (let ((bindings (derive-bindings-from-ll lambda-list)))
      (dolist (form body)
        (walk form (make-env bindings env) (cons form call-stack)))
      (dolist (bind bindings)
        (unless (used bind)
          (format t "unused: ~A: ~A~%" var call-stack))))))

```

Figure 5: Handling the (quote ...) and (lambda ...) special forms.

whose first elements are `quote` or `lambda` are special and must be intercepted before the walker reaches the `quote` and `lambda` symbols themselves.

The `cons` specializer provides a mechanism for making a method applicable for such a list. Figure 5 implements methods for handling `quote` and `lambda` forms. The first method is applicable if its first argument is a list whose first element is the symbol `quote`. Since an evaluator would simply return the second element of this special form unevaluated, there can be no variable references inside it; so the code walker simply returns `nil`.

The second method handles `lambda` forms by creating new bindings as indicated by the lambda list and walking the body of the `lambda` with those bindings in place. After the code walker returns from walking the lambda body we can report if any of the new bindings were not referenced by the walked code.⁵

This implementation of `walk` is a simplified version of a walker for `SKILL` that is used in production; we have elided many details of the full version. For

section 4.2.

⁵ The implementations of the functions `derive-bindings-from-ll` and `make-env` are omitted for this illustration as they do not aid in understanding extensible specializers. The `derive-bindings-from-ll` function returns a list of *binding* objects from a lambda list. The `make-env` function allocates a new *environment* which references the given list of binding objects, and also references the given parent environment.

example, rather than printing diagnostics, the walker communicates with the environment, allowing the offending forms to be highlighted in the editor; additionally, the walker supports a much broader range of the SKILL language semantics, including ignorable and global variables, assignment, macro expansion and more special forms. The user-defined `cons` specializer presented here allows us to have a single generic function, `walk`, whose methods specialize on all of the different types of forms that must be handled differently.

As an example of perhaps a potentially generally useful specializer type, consider a specializer corresponding to a pattern, similar to those found in the ML family of languages. Using the mechanisms presented in this paper, it is possible to have the dispatch over patterns optimized as is expected in those languages, while still retaining the customary run-time extensibility of Lisp, by lazily compiling the dispatch (using algorithms such as those in [LFM01]) and invalidating the compiled code if methods are added or removed to the pattern-matching generic function.

An application which, we believe, would benefit from a protocol for defining specializers for which there is no corresponding hierarchy is an Emacs-like text editor, where ‘minor modes’ can affect the functionality of keystrokes and editor function calls. For instance, in the Climacs text editor [RSM05], minor modes are currently implemented by the creation of anonymous classes with a combination of superclasses corresponding to the currently-active modes, whereas it should be simpler to express this as a dispatch on aspects of the current editor state.

4 Implementation Details

4.1 Skill, SKILL++ and VCLOS

To address the limitations of SKILL++ (see section 2.1) a new object system for SKILL was needed, to provide more of the features of CLOS. The new object system was required to be able to interface to programs written in the existing SKILL++ system, and allow object-oriented techniques to be used on existing systems whose object models are not changeable, while also being extensible for the types of problems faced in application programming for IC development.

Neither VCAD (an organizational department within Cadence Design Systems) nor VCAD’s customers have write access to the SKILL implementation, and so the language itself cannot be changed: the object-oriented extension must be provided as a loadable SKILL application. From its Lisp heritage, SKILL can be altered in this way so that the extension seems native to the SKILL programmer and invisible to the end-user.

4.1.1 VCLOS and its Metaobject Protocol

The resulting system, VCAD CL-like Object System (VCLOS), was developed over several years; the major difference from CLOS and its Metaobject Protocol [KdRB91] is that more importance is given to the `mop:specializer` metaobject class, rather than having most of the dispatch functionality of generic functions be computed from the `class` of arguments.

The VCLOS Metaobject Protocol implemented is then similar to the CLOS MOP, with the following points to note:

- the `ClosClassSpecializer` and `ClosEqvSpecializer` classes are both subclasses of `ClosSpecializer`, while users are encouraged to define their own subclasses of `ClosSpecializer` by the provision of a protocol for using them in computation of the effective method (described further below);
- in VCLOS, `ClosComputeApplicableMethodsUsingSpecializers` takes the place of `mop:compute-applicable-methods-using-classes` in the standard AMOP generic function invocation protocol;
- a good CLOS implementation will memoize the results of `mop:compute-applicable-methods-using-classes` if possible, with a key based on the classes of the arguments (see [KR93] for some details). VCLOS supports memoization based on specializer names, computed using `ClosComputeSpecializerNames`.

In order to use a user-defined specializer class, the user must define a subclass of `ClosSpecGenericFunction`, the generic function subclass following the protocols for extensible specializers. The protocol defined on `ClosSpecGenericFunction` allows for the user to specify how to put specializers in precedence order through defining methods on Metaobject Protocol functions: `ClosAvailableSpecializers` and `ClosCmpLikeSpecializers`. The method on `ClosAvailableSpecializers` applicable to a particular generic function class must return a list of specializer class names, from most specific to least specific; methods on `ClosCmpLikeSpecializers` must decide which of two specializers of the same class (assumed both applicable to the same generic function argument) is more specific.

Among the other Metaobject Protocol functions which need to have methods defined for user-defined specializers to work are `ClosArgMatchesSpecializerP`, a function of a specializer and an arbitrary object, which returns true if a specializer corresponds to a type of which the given object is a member, and `ClosGetClassPrecedenceList` (which should perhaps have been called `ClosGetSpecializerPrecedenceList`), which for a given specializer computes a linearization of its less-specific specializers.

The treatment mapping specializer surface syntax such as `(cons quote)` to specializer metaobjects is performed by generic functions `ClosMatchesSpecializerSyntaxP` and `ClosSetSpecializerData`. Note that in this respect the SKILL protocol and the extension to the Common Lisp Metaobject Protocol described in appendix A differ in the approach taken, as in Common Lisp the specializer syntax is sensitive to the lexical environment.

The SKILL implementation of VCLOS provides memoization, keyed on the specializers of the arguments, to the effective method, allowing the elision of calls to `ClosComputeApplicableMethodsUsingSpecializers`, in a similar way to the CLOS MOP protocol around `mop:compute-applicable-methods-using-classes`. There is still overhead involved in computing appropriate specializers corresponding to the arguments, relative to the baseline of computing an argument's class, but this memoization can significantly reduce the overhead of using a non-standard specializer.

4.2 Common Lisp and the Metaobject Protocol

Much of the work in implementing custom specializers in SKILL was of course taken up by providing a suitably rich object system such that customizations can meaningfully be made: essentially, taking a single-dispatch, single-inheritance object system as found in SKILL++ and implementing on top of it a multiple-dispatch, multiple-inheritance system with a Metaobject Protocol. By contrast, in Common Lisp (with the *de facto* standard MOP) we already have most of the framework for the implementation of custom specializers; for basic operation, we only require a few non-standard operators.

The Lisp-like language we have used for our examples, and the actual implementation of the specializer metaobject class in SKILL, share one important difference in detail from Common Lisp. In Common Lisp's `defmethod` macro, the `eq1` specializer specifies not a specialization on a following literal, but instead a specialization on the value of a form in the lexical environment of the method definition.

This detail implies that there must be an operator, similar to `mop:make-method-lambda`, which is capable of converting surface syntax such as `(eq1 foo)` into code which constructs an `mop:eq1-specializer` metaobject at the time when the `defmethod` is executed. Of course, we could restrict the use of the lexical environment to the standardized `eq1` specializer, but since it is possible to support culturally-compatible use of the lexical environment through a relatively straightforward backward-compatible extension to the CLOS Metaobject Protocol (see appendix A), we choose to do so, defining our new operator as `make-method-specializers-form`. For convenience, we also suggest `parse-specializer-using-class` and `unparse-specializer-using-class` to con-

sume and produce user-friendly representations of specializers, for use in `find-method` and printed representations of methods.

4.2.1 VCLOS implementation in SBCL

We have in addition implemented a version of the SKILL and VCLOS Metaobject Protocol described in section 4.1.1 above, and used it to run the `walk` example from section 3. The implementation of the VCLOS protocol in SBCL's MOP is by no means complete and certainly not industrial-strength; however, even the simple implementation raises some issues.

Firstly, initial explorations revealed that current Common Lisp implementations have only partial support for subclassing `mop:specializer`; most implementations will allow defining the subclass, but very few recognize such a subclass as a valid specializer. In the implementation for SBCL, we had to alter a number of places in the CLOS implementation where the assumption had been made that a specializer was either a `class` or an `eql-specializer`.

Secondly, since the system needs to call, as part of the discriminating function, a new function `compute-applicable-methods-using-specializers` instead of the usual `compute-applicable-methods-using-classes` (and we need to be calling `specializer-of` rather than `class-of` on the generic function arguments), we must override `mop:compute-discriminating-function` for our generic function class. This in turn means that we need to interpret or compile the result of `mop:compute-effective-method` ourselves, which is not a straightforward procedure, as suitable definitions for `call-method` and `make-method` need to be provided; `mop:compute-effective-method` returns a form, not something which is directly executable.

Additionally, we need to provide an implementation of `compute-applicable-methods`, as well as the new protocol function `compute-applicable-methods-using-specializers`, because the new methods must call our protocol function `specializer-applicable-p` (for determining whether an argument matches a specializer). An implementation is not difficult in principle, but tedious and error-prone; because of limited resources we have instead provided a method which considers only the first required argument to a generic function, leaving the implementation of the multiple-dispatch aspect for further work.

While the presence of user-defined specializers makes it harder to reason about the cacheability of effective methods or lists of applicable methods, there are still points in the protocol discussed above which would allow a value to be computed once and reused for efficiency; our current implementation in Common Lisp does not take advantage of these.

5 Conclusions and Future Work

We have presented the implementation and use of custom specializers both in a Lisp dialect where that functionality is used, and also in Common Lisp, a language with a standardized core and *de facto* standard Metaobject Protocol.

The implementation in SKILL is complete and used in production: the implementation is fully functional, has an extensive suite of unit tests, and is part of live design projects. Much time has been spent on optimization and refactoring for performance and readability of the code, but of course much more work in this area could be done.

The functionality for the user to define their own specializer classes has been available in SBCL since May 2007; in practice the design space seems to be too general for easy exploration: having to reimplement the entirety of `compute-applicable-methods` and `mop:compute-applicable-methods-using-classes` is excessive. Our ‘toy’ implementation of the VCLOS protocols should be refined and extended, so that users can experiment with their specializer classes without having to reimplement complicated protocol functions.

In particular, it is important to take advantage of the various points in the protocol where memoization can be used (in the calculation of the effective method, for instance), so that the run-time overhead from use of user-defined specializers is as low as possible. Doing this would allow us to compare the efficiency of the protocol implementation in SKILL and Common Lisp, and to identify further points for optimization if necessary.

One thing missing from the Metaobject Protocol for Common Lisp (including our extension) is a general case for something that SBCL in particular takes advantage of: in SBCL, a method definition with a standard specializer will inform the method body (by inserting a declaration) that the corresponding element in the method function arguments is of a relevant type. There is at present no way of communicating this information for an arbitrary user-defined specializer.

Acknowledgments

SKILL® is a registered trademark of Cadence Design Systems, Inc.

References

- [Bar90] Timothy J. Barnes. SKILL: A CAD system extension language. In *DAC '90*, pages 266–271. ACM, 1990.
- [BdL00] Tim Bradshaw and Raymond de Lacaze. A Survey of Current CLOS MOP Implementations. In *Japan Lisp Users Group Meeting*, 2000.
- [BKK⁺86] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. Common Loops: Merging Lisp and Object-Oriented Programming. In *OOPSLA '86 Proceedings*, pages 17–29, 1986.

- [KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
- [KR93] Gregor Kiczales and Luis H. Rodriguez Jr. Efficient method dispatch in PCL. In Andreas Paepcke, editor, *Object-Oriented Programming: the CLOS Perspective*, pages 335–348. MIT Press, Cambridge, Mass., 1993.
- [LFM01] Fabrice Le Fessant and Luc Maranget. Optimizing Pattern Matching. In *ICFP'01 Proceedings*, pages 26–37, 2001.
- [Mil04] Todd Milstein. Practical Predicate Dispatch. In *OOPSLA '04*, pages 345–364. ACM, 2004.
- [Moo86] David Moon. Object Oriented Programming with *Flavors*. In *OOPSLA'86 Proceedings*, pages 1–8, 1986.
- [N⁺00] William Harold Newman et al. SBCL User Manual. <http://www.sbcl.org/manual/>, 2000.
- [PC94] Kent Pitman and Kathy Chapman, editors. *Information Technology – Programming Language – Common Lisp*. Number 226–1994 in INCITS. ANSI, 1994.
- [Pet93] Edwin S. Petrus. SKILL: a Lisp based extension language. *Lisp Pointers*, VI(3):71–79, 1993.
- [RSM05] Christophe Rhodes, Robert Strandh, and Brian Mastenbrook. Syntax Analysis in the Climacs Text Editor. In *International Lisp Conference Proceedings*, 2005.
- [Ste90] Guy L. Steele, Jr. *Common Lisp: The Language*. Digital Press, second edition, 1990.
- [Uck01] Aaron Mark Ucko. Predicate dispatching in the Common Lisp Object System. Technical Report AITR-2001-006, MIT AI Lab, Cambridge, MA, 2001. MEng thesis.

A Common Lisp extension to the MOP

The CLOS Metaobject Protocol requires little extension to support everything discussed in this paper. On a fundamental level, in fact, no new operators are required, though for convenient use of the standardized operators `defmethod` and `find-method` we propose an analogue to `mop:make-method-lambda` and operators to parse and unparse parameter specializer names.

In order to emulate the specific specializer handling present in VCLOS, an overriding implementation of `compute-applicable-methods` and `mop:compute-applicable-methods-using-classes` would be necessary. However, for any particular strategy for dealing with the method applicability and ordering computation, such an implementation need only be written once; once written, the CLOS user would be free to implement specializers using the defined protocol.

A.1 Dictionary

Generic Function `parse-specializer-using-class`

Syntax:

`parse-specializer-using-class` *generic-function* *specializer-name*

This generic function returns an instance of `mop:specializer`, representing the specializer named by *specializer-name* in the context of *generic-function*.

Primary Method `parse-specializer-using-class` (*gf* `standard-generic-function`) (*name* *t*)

This method applies the standard parsing rules for consistency with the specified behaviour of `find-method`.

Generic Function `unparse-specializer-using-class`

Syntax:

`unparse-specializer-using-class` *generic-function* *specializer*

This generic function returns the name of *specializer* for generic functions with class the same as *generic-function*

Primary Method `unparse-specializer-using-class` (*gf* `standard-generic-function`) (*specializer* *specializer*)

This method applies the standard unparsing rules for consistency with the specified behaviour of `find-method`.

Generic Function `make-method-specializers-form`

Syntax:

`make-method-specializers-form` *generic-function* *method* *specializer-names* *env*

This function is called with (maybe uninitialized, as with the analogous arguments to `mop:make-method-lambda`) *generic-function* and *method*, and a list of specializer names (being the parameter specializer names from a `defmethod` form, or the symbol `t` if unsupplied), and returns a form which evaluates to a list of specializer objects in the lexical environment of the `defmethod` form.

Primary Method `make-method-specializers-form` (*gf* `standard-generic-function`) (*method* `standard-method`) *names env*

This method implements the standard behaviour for parameter specializer names.

Binary Methods Programming: the CLOS Perspective

Didier Verna

(EPITA Research and Development Laboratory, Paris, France

didier@lrde.epita.fr)

Abstract: Implementing binary methods in traditional object-oriented languages is difficult: numerous problems arise regarding the relationship between types and classes in the context of inheritance, or the need for privileged access to the internal representation of objects. Most of these problems occur in the context of statically typed languages that lack multi-methods (polymorphism on multiple arguments). The purpose of this paper is twofold: first, we show why some of these problems are either non-issues, or easily solved in Common Lisp. Then, we demonstrate how the Common Lisp Object System (CLOS) allows us not only to implement binary methods in a straightforward way, but also to support the concept directly, and even enforce it at different levels (usage and implementation).

Key Words: Binary methods, Common Lisp, object orientation, meta-programming
Category: D.1.5, D.3.3

1 Introduction

Binary operations work on two arguments of the same type. Common examples include arithmetic operations ($=$, $+$, $-$ *etc.*) and ordering relations ($=$, $<$, $>$ *etc.*). In the context of object-oriented programming, it is often desirable to implement binary operations as methods applied on two objects of the same class in order to benefit from polymorphism. Such methods are hence called “binary methods”.

Implementing binary methods in many traditional object-oriented languages is a difficult task: the relationship between types and classes in the context of inheritance and the need for privileged access to the internal representation of objects are the two most prominent problems. In this paper, we approach the concept of binary method from the perspective of Common Lisp.

The paper is composed of two main parts. In section 2, we show how two problems mentioned above are either non-issues, or easily solved. In section 3, we show how to support the concept of binary methods *directly* into the language, and demonstrate how to ensure not only a correct *usage* of it, but also a correct *implementation* of it.

2 Binary methods non-issues

In this section, we describe the two major problems with binary methods in a traditional object-oriented context, as pointed out by [Bruce et al., 1995]: mixing

types and classes within an inheritance scheme, and the need for privileged access to the internal representation of objects. We show why the former is a non-issue in Common Lisp, and how the latter can be solved. In order to illustrate our matter, we take the same examples as used by [Bruce et al., 1995], and provide excerpts from a sample implementation in C++ for comparison.

2.1 Types, classes, inheritance

Consider a `Point` class representing 2D points from an image, equipped with an equality operation. Consider further a `ColorPoint` class representing a `Point` associated with a color. A natural implementation would be to inherit from the `Point` class, as shown in listing 1 (C++ version, details omitted).

```

class Point
{
    int x, y;

    bool equal (Point& p)
    { return x == p.x && y == p.y; }
};

class ColorPoint : public Point
{
    std::string color;

    bool equal (ColorPoint& cp)
    { return color == cp.color
      && Point::equal (cp);
    }
};

```

Listing 1: Excerpt from the `Point` class

However, this implementation does not behave as expected because what we have done in the `ColorPoint` class is simply *overload* the `equal` method: `ColorPoint` objects manipulated as `Point` ones will only see the definition for `equal` from the base class, as demonstrated in listing 2.

```

bool foo (Point& p1, Point& p2)
{
    // Point::equal is called
    return p1.equal (p2);
}

ColorPoint p1 (1, 2, "red");
ColorPoint p2 (1, 2, "blue");
foo (p1, p2); // => true. Wrong!

```

Listing 2: Method overloading

In order to find the proper method definition at run-time in C++, one needs *virtual methods* (obtained by simply prefixing the methods declarations in figure 1 with the keyword `virtual`). Unfortunately, such an implementation doesn't behave as expected. Indeed, C++ does not allow the arguments of a virtual

method to change type as in figure 1, because this would not statically type check.

2.1.1 The static type safety problem

By definition of inheritance, a `ColorPoint` *is* a `Point`, so it should be possible to use a `ColorPoint` where a `Point` is expected. Consider the situation described in listing 3. The function `foo` expects two `Point` arguments, but actually gets a `ColorPoint` as the first one. Assuming that the `equal` method from the *exact* class is called (hence `ColorPoint::equal`), we see that this method could try to access the `color` field in a `Point`, which does not exist. Therefore, if we want to preserve static type safety, this code should not compile.

```
bool foo (Point& cp, Point& p)           ColorPoint cp (1, 2, "red");
{                                       Point      p (1, 2);
  return cp.equal (p);
}                                       foo (cp, p); // => ???
```

Listing 3: The static type safety problem

In order to prevent this situation from happening, we see that the `ColorPoint::equal` method should not expect to get anything more specific than a `Point` object. More precisely, maintaining static type safety in a context of inheritance implies that polymorphic methods must follow a *contravariance* [Castagna, 1995] rule on their arguments: a derived method in a subclass can be prototyped as accepting arguments of the same class or of a superclass of the original arguments only.

2.1.2 A non-issue in Common Lisp

In languages such as C++, methods belong to classes and the polymorphic dispatch depends only on one parameter: the class of the object through which the method is called. The Common Lisp Object System (CLOS [Keene, 1989]), on the other hand, differs in two important ways.

1. Firstly, methods do not belong to classes: a polymorphic call *appears* in the code like an ordinary function call. Functions the behavior of which are provided by such methods are called *generic functions*.
2. Secondly, and more importantly, CLOS supports *multi-methods*, that is, polymorphic dispatch based on any number of arguments, not only the first one (**this** in C++).

```

(defclass point ()
  ((x :reader point-x)
   (y :reader point-y)))

(defclass color-point (point)
  ((color :reader point-color)))

(defmethod point=
  ((a point) (b point))
  (and (= (point-x a) (point-x b))
        (= (point-y a) (point-y b))))

(defmethod point=
  ((a color-point) (b color-point))
  (and (string= (point-color a)
                (point-color b))
        (call-next-method)))

```

Listing 4: The Point hierarchy in Common Lisp

In order to clarify this, listing 4 provides a sample implementation of the `Point` hierarchy in Common Lisp (details omitted). As you can see, `point` and `color-point` classes are defined with only data members (called *slots* in the CLOS jargon). Instead of being class members, two methods on the *generic function* `point=` are defined by calls to `defmethod`. As you can see, a special argument syntax lets you specify the expected class of each: we provide a method for comparing two `point` objects, and one for comparing two `color-point` ones. Testing for equality between two points is now simply a matter of calling the generic function as follows:

```
(point= p1 p2)
```

According to the *exact* classes of *both* of the objects, the correct method is used. The case where the generic function would be called with two arguments of different classes (for example, `point` and `color-point`) will be addressed in section 3.2.

2.2 Privileged access to objects internals

The second problem exposed by [Bruce et al., 1995] involves more complex situations in which the need for accessing the objects internals (normally hidden from public view) is required.

Consider a type `IntegerSet`, representing sets of integers, with an interface providing methods such as the following (their purpose should be obvious):

```
add    (i: Integer): Unit
member (i: Integer): Boolean
```

and also a binary method like the one below:

```
superSet (a: IntegerSet, b: IntegerSet): Boolean
```

Consider further that several implementations are available (for instance, for efficiency reasons), effectively storing the set as a list or array of integers, as a

bitstring or whatever else. While implementing `add` and `member` is not an issue at all, the binary method *is* problematic. Indeed, this method needs to access the individual elements of the sets. It is possible to enrich the above interface with a method returning the sets elements in a single format (for instance, a list), but the concern expressed by [Bruce et al., 1995] is that it might be preferable to work directly on the internal representation for efficiency reasons. The conclusion they draw from this example is twofold:

1. a mechanism is needed for constraining both arguments of the binary method to be not only of the same type, but also of the same implementation,
2. another mechanism is also needed to allow access to this internal representation while keeping it hidden from general public view.

2.2.1 Types vs. implementation

It would be slightly abusive to claim that point 1 above is a “non-issue” in Common Lisp because the question does not arise exactly in the same terms. When considering constraining both type and implementation to be the same, the authors are silently assuming that there is (or should be) a clear distinction between them. As a matter of fact, CLOS does not explicitly provide any such distinction.

In dynamic languages such as Common Lisp however, we might think of solutions in which this distinction is *intentionally* blurred. For instance, we can define a *single integer-set* class equipped with a `set` slot, and let different instances of this class use different `set` types (lists, arrays, bitstrings *etc.*) at run-time. In such a case, the `super-set` function need not be generic anymore (since we have only one class to deal with), but will in turn involve a generic call to effectively compare sets, once their actual type is known.

Also, note that contrary to the first conclusion drawn by [Bruce et al., 1995], the multiple dispatch offered by Common Lisp generic functions will allow us to implement this comparison even for different kinds of sets (however, this cannot be considered a “binary” operation anymore).

2.2.2 Data encapsulation

The last problem we have to address is the need for accessing the internal representation of objects while still following the general principle of information hiding. The assumption is that in the general case, only the type (or the interface) of an object should be public. Common Lisp itself does not provide any functionality for data encapsulation, but the *package* system is perfectly suited to this task.

Back to our original example (the `point` class), we now roughly describe how one would use the package system to perform implementation hiding. Many important aspects of Common Lisp packages are omitted because our point is not to describe them thoroughly.

```
(defpackage :point                                     (in-package :point)
  (:use :cl)                                          (defclass point ()
    (:export :point                                   ((x :reader point-x)
          :point-x                                    (y :reader point-y)))
          :point-y))
```

Listing 5: The point class package

The right side of listing 5 shows a definition of the `point` class, which is no different from the one in listing 4; there is nothing in the class definition to separate interface from implementation. Only the first line of code is new: it merely tells Common Lisp that the current package should be a certain one named `point`. When Common Lisp encounters, at read-time, a name for a symbol which is not found, it automatically creates the corresponding symbol and adds it to the current package. In our case, the effect is to add 5 new symbols into the `point` package: `point`, `x`, `y`, `point-x` and `point-y`. Note that we are only talking about *symbols* here. Associated variables or functions do *not* belong to packages.

In order to effectively declare what is “public” and “private” in a package, one has to provide a package definition such as the one depicted on the left side of listing 5. The `:use` clause specifies that while in the `point` package, all public symbols from the `cl` package (the one that provides the Common Lisp standard) are also directly accessible. Consider that if this clause had been missing, we could not have accessed the macro definition associated with the symbol `defclass`. The `:export` clause specifies which symbols are public. As you can see, the class name and the accessors are made so, but the slot names remain private.

Now, in order to access the public (exported) symbols of the `point` package, one has two options. The first one is to use symbol names *qualified* by the package name, such as `point:point-x`. The second option is to `:use` the package, in which case all exported symbols become directly accessible, without any qualification. Hence, the `point=` method in listing 4 can be used as-is.

As for the question of accessing private information, this is where the surprise is the most striking for people accustomed to other package systems or information hiding mechanisms: any private (not exported) symbol from a package can be accessed with a double-colon qualified name from anywhere. Thus, one

could access the slot values in the `point` class at any place in the code using the symbol names `point::x` and `point::y`.

Accessing a package's private symbols should be considered bad programming style, and used with caution because it effectively breaks modularity. But it is nevertheless easy to do so, and although maybe surprising, is typical of the Lisp philosophy: be very permissive in the language and put more trust on the programmer's skills.

One important design consideration here is that the package system and the object-oriented layer are completely orthogonal: compare this with languages such as C++ in which information hiding is done by the object-oriented layer itself (`public`, `protected` and `private` members). Also, note that no additional mechanism is needed for privileged access either. One simply uses an additional colon when one really wants to. Again, compare this with languages such as C++ in which an additional machinery is needed (`friend` functions, methods or classes).

For the record, note that Common Lisp allows for completely hiding symbols (they are said to be *uninterned*), but doing that is definitely not the “Lisp way”.

3 Binary methods enforcement

While the previous section demonstrated how straightforward it is to implement binary methods, there is no explicit support for them in the language. In the remainder of this paper, we gradually add support for the concept *itself*, thanks to the expressiveness of CLOS and the flexibility of the CLOS Meta-Object Protocol (MOP). From now on, we will use the term “binary function” as a shorthand for “binary generic function”.

3.1 Method combinations

When calling `point=` with two `color-point` objects, both of the methods we defined are applicable because a `color-point` object can be seen as a `point` one. More generally, for each generic function call, several methods might fit the classes of the arguments. These methods are called “applicable methods”.

When a generic function is called, CLOS computes the list of applicable methods and sorts it from the most to the least specific one. Within the body of a method, a call to `call-next-method` triggers the execution of the next most specific applicable method. In our example (listing 4), when calling `point=` with two `color-point` objects, the most specific method is the second one, which specializes on the `color-point` class, and `call-next-method` within it triggers the execution of the other, hence completing the equality test (this is roughly the equivalent of calling `Point::equal` in the C++ version).

An interesting feature of CLOS is that, contrary to other object-oriented languages where only one method is applied (this is also the default behavior in CLOS), it is possible to use all, or some of the applicable methods to form the global execution of the generic function (resulting in what is called an *effective method*).

This concept is known as *method combination*: a way to combine the results of all or some of the applicable methods in order to form the result of the generic function call itself. CLOS provides several predefined method combinations, as well as the possibility for the programmer to define his own.

In our example, one particular (predefined, for that matter) method combination is of interest to us: our equality concept is actually defined as the logical *and* of all local equalities in each class. Indeed, two `color-point` objects are equal if their `color-point`-specific parts are equal *and* if their `point`-specific parts are also equal.

This can be directly implemented by using the `and` method combination, as shown in listing 6.

```
(defgeneric point= (a b)
  (:method-combination and))
  (defmethod point= and
    ((a point) (b point))
    (and (= (point-x a) (point-x b))
          (= (point-y a) (point-y b))))
  (defmethod point= and
    ((a color-point) (b color-point))
    (string= (point-color a) (point-color b)))
```

Listing 6: The `and` method combination

As you can see, the call to `defgeneric` (otherwise optional) specifies the method combination we want to use, and both calls to `defmethod` are modified accordingly. The advantage of this new scheme is that each method can now concentrate on the local behavior only: there is no more call to `(call-next-method)`, as the logical *and* combination is performed automatically. This also has the advantage of preventing possible bugs resulting from an unintentional omission of this very same call.

Note that what we have done here is actually modify the semantics of the dynamic dispatch mechanism. While other object-oriented languages offer one single, hard-wired, dispatch procedure, CLOS lets you (re)program it.

3.2 Enforcing a correct usage of binary functions

In this section, we start providing explicit support for the concept of binary function itself by addressing another problem from our previous implementa-

tion. Our equality concept requires that only two objects of the same exact class be compared. However, nothing prevents one from using the `point=` binary function for comparing a `color-point` with a `point` for instance. Our current implementation of `point=` is unsafe because such a comparison is perfectly valid code and the error would go unnoticed. Indeed, since a `color-point` *is* a `point` by definition of inheritance, the first specialization (the one on the `point` class) *is* an applicable method, so the comparison will work, but only check for point coordinates.

3.2.1 Introspection in CLOS

We can solve this problem by using the introspection capabilities of CLOS: it is possible to retrieve the class of a particular object at run-time. Consequently, it is also very simple to check that two objects have the same exact class, and trigger an error otherwise. In listing 7, we show a new implementation of `point=` making use of the function `class-of` to retrieve the exact class of an object, in order to perform such a check.

```
(defmethod point= and ((a point) (b point))
  (assert (eq (class-of a) (class-of b)))
  (and (= (point-x a) (point-x b))
        (= (point-y a) (point-y b))))
```

Listing 7: Introspection example in CLOS

We chose to perform this check only in the least specific method in order to avoid code duplication, because we know that this method will be used for all `point` objects, including instances of subclasses. One drawback of this approach is that since this method is always called last, it is a bit unsatisfactory to perform the check in the end, after all more specific methods have been applied, possibly for nothing.

3.2.2 Before-methods

CLOS has a feature perfectly suited to (actually, even designed for) this kind of problem. The methods we have seen so far are called *primary methods*. They resemble methods from traditional object-oriented languages (with the exception that they can be combined together). CLOS also provides other kinds of methods, such as *before* and *after-methods*. As their name suggests, these methods are executed *before* or *after* the primary ones, and are typically used for side-effects.

Unfortunately, before and after-methods cannot be used with the `and` method combination described in section 3.1. Thus, assuming that we are back to the

initial implementation described in listing 4, listing 8 demonstrates how to properly place the check for class equality. Note the presence of the `:before` keyword in the method definition.

```
(defmethod point= ((a point) (b point)) :before
  (assert (eq (class-of a) (class-of b))))
```

Listing 8: Using before-methods

We want this check to be performed for all `point` objects, including instances of subclasses, so this method is specialized only on `point`, and hence applicable to the whole potential `point` hierarchy. But note that even when passing `color-point` objects to `point=`, the before-method is executed before the primary ones, so an occasional usage error is signaled as soon as possible. This scheme effectively removes the need to perform the check in the first method itself, which is much cleaner at the conceptual level.

3.2.3 A meta-class for binary functions

There is still something conceptually wrong with the solutions proposed in the previous sections: the fact that it makes no sense to compare objects of different classes belongs to the concept of binary function itself, not to the `point=` operation. In other words, if we ever add a new binary function to the `point` hierarchy, we don't want to duplicate the code from listings 7 or 8 yet again.

What we really need is to be able to express the concept of binary function directly. A binary function *is* a generic function with a special, constrained behavior (taking only two arguments of the same class). In other words, it is a *specialization* of the general concept of generic function. This strongly suggests an object-oriented model, in which binary functions are subclasses of generic functions. This conceptual model happens to be accessible if we delve a bit more into the CLOS internals.

CLOS itself is written on top of a *Meta Object Protocol*, simply known as the CLOS MOP [Paepcke, 1993, Kiczales et al., 1991], which architects CLOS itself in an object-oriented fashion: classes (the result of calling `defclass`) are CLOS (meta-)objects, that is, instances of certain (meta-)classes. Similarly, a user-defined generic function (the result of calling `defgeneric`) is a CLOS object of class `standard-generic-function`. We are hence able to implement binary functions by subclassing standard generic functions, as shown in listing 9.

The `binary-function` class is defined as a subclass of `standard-generic-function`, and does not provide any additional slot. Since

```

(defclass binary-function (standard-generic-function)
  ()
  (:metaclass funcallable-standard-class))

(defmacro defbinary (function-name lambda-list &rest options)
  (when (assoc ':generic-function-class options)
    (error
     " :generic-function-class option prohibited" ))
  `(defgeneric ,function-name ,lambda-list
    (:generic-function-class binary-function)
    ,@options))

```

Listing 9: The binary function class

instances of this class are meant to be called as functions, it is also required to state that the `binary-function` meta-class (the class of the `binary-function` class meta-object) is a “funcallable” meta-object. This is done through the `:metaclass` option, which is given `funcallable-standard-class` and not just `standard-class`.

Now that we have a proper meta-class for binary functions, we need to make sure that our binary generic functions are instantiated from it. Normally, one specifies the class of newly created generic functions by passing a `:generic-function-class` argument to `defgeneric`. If this argument is omitted, generic functions are instantiated from the `standard-generic-function` class. With a few lines of macrology, we make this process easier by providing a `defbinary` macro that is to be used instead of `defgeneric`. This macro is designed as a syntactic clone of `defgeneric`, but we could also think of all sorts of modifications, including enforcing the lambda-list (the generic call prototype) to be of exactly two arguments *etc.*

3.2.4 Back to introspection

Now that we have an explicit implementation of the binary function concept, let us get back to our original problem: how and when can we check that only points of the same class are compared ?

For each generic function call, we saw that CLOS must calculate the sorted list of applicable methods for this particular call. In most cases, this can be figured out from the classes of the arguments to the generic call. The CLOS MOP implements this by calling `compute-applicable-methods-using-classes` (`c-a-m-u-c` for short).

`c-a-m-u-c` is not an ordinary function, but a *generic* one, taking two arguments: first, the generic function meta-object involved in the call (in our case, that would be the `point=` one created by the call to `defgeneric`), and the list of the arguments classes involved in the generic call (in our case, that would be a list of two element, either `point` or `color-point` class meta-objects).

```
(defmethod c-a-m-u-c :before ((bf binary-function) classes)
  (assert (apply #'eq classes)))
```

Listing 10: Back to introspection

This generic function is interesting to us because, conceptually speaking, before even calculating the applicable methods given the arguments `classes`, we should make sure that these two classes are the same. This strongly suggests a specialization with a before-method (see section 3.2.2), and this is demonstrated in listing 10. As you can see, this new method only applies to binary functions, thanks to the specialization of its first argument on the `binary-function` class. The advantage is that the check now belongs to binary function concept itself, and not anymore to each individual function one might want to implement.

3.3 Enforcing a correct implementation of binary functions

In the previous section, we have made sure that binary functions are *used* as intended, and we have made that part of their implementation. In this section, we make sure that binary functions are *implemented* as intended, and we also make this requirement part of their implementation.

3.3.1 Properly defined methods

Just as it makes no sense to compare points of different classes, it makes even less sense to *implement* methods to do so. The CLOS MOP is expressive enough to make it possible to implement this constraint directly.

When a call to `defmethod` is issued, CLOS must register the new method into the concerned generic function. This is done in the MOP through a call to `add-method`. It is not an ordinary function, but a *generic* one, taking two arguments: first, the generic function meta-object involved in the call (in our case, that would be the `point=` one created by the call to `defgeneric`), and the newly created method object.

This generic function is interesting to us because, conceptually speaking, before registering the new method, we should make sure that it specializes on two identical classes. This strongly suggests a specialization with a before-method (see section 3.2.2), and this is demonstrated in listing 11.

Again, this new method only applies to binary functions, thanks to the specialization of its first argument on the `binary-function` class. And again, the advantage is that the check belongs directly to the binary function concept itself, and not to every individual function one might want to implement. The function `method-specializers` returns the list of argument specializations from

```
(defmethod add-method :before ((bf binary-function) method)
  (assert (apply #'eq (method-specializers method))))
```

Listing 11: Binary method definition check

the method's prototype. In our examples, that would be `(point point)` or `(color-point color-point)`, so all we have to do is check that the members of this list are actually the same.

3.3.2 Binary completeness

One might realize that our `point=` concept is not yet completely enforced, if for instance, the programmer forgets to implement the `color-point` specialization: when comparing to points at the same coordinates but with different colors, only the coordinates would be checked and the test would silently yet mistakenly succeed. It would be an interesting safety measure to ensure that for each defined subclass of the `point` class, there is also a corresponding specialization of the `point=` function (we call that *binary completeness*), and it should be no surprise that the CLOS MOP lets you do just that.

Remember that the function `c-a-m-u-c` is used to sort out the list of applicable methods. Again, this is very interesting to us because the check for binary completeness involves introspection on exactly this list (to see if some methods are missing). What we can do is thus specialize on the *primary* method this time, retrieve the list in question simply by calling `call-next-method`, and then do our own work, as depicted in listing 12. The built-in `c-a-m-u-c` returns two values, the first of which is the list of applicable methods. After we perform our check for completeness (and possibly trigger an error), we simply return the values we got from the default method.

```
(defmethod c-a-m-u-c ((bf binary-function) classes)
  (multiple-value-bind (methods ok) (call-next-method)
    (when ok
      ;; Check for binary completeness
    )
    (values methods ok)))
```

Listing 12: Binary completeness skeleton

Our check involves two different things: first we have to assert that there exists a specialization for the exact classes of the objects we are comparing (otherwise, as previously mentioned, a missing specialization for `color-point`

would go unnoticed). This is demonstrated in listing 13. The most specialized applicable method is the first one in the list. The classes on which it specializes are retrieved by calling `method-specializers` (it suffices to retrieve the first one because we already know that both are identical; see listing 11). We then check that the classes of the arguments involved in the generic call (the `classes` parameter) match the most specific specialization.

```
(let* ((method (car methods))
      (class (car (method-specializers method))))
  (assert (equal (list class class) classes))
  ;; ...
```

Listing 13: Binary completeness check n.1

Next, we have to check that the whole super-hierarchy has properly specialized methods (none were forgotten). This is demonstrated in listing 14. We define a local recursive function `find-binary-method` that we first apply on the bottommost class in the hierarchy we are checking (the `class` binding from listing 13).

```
;; ...
(labels
  ((find-binary-method (class)
    (find-method bf (method-qualifiers method) (list class class))
    (dolist
      (cls (remove-if
            #'(lambda (elt) (eq elt (find-class 'standard-object)))
            (class-direct-superclasses class)))
      (find-binary-method cls))))
  (find-binary-method class))
```

Listing 14: Binary completeness check n.2

The function `find-method` retrieves a method meta-object for a particular generic function satisfying a set of qualifiers and a set of specializers. In our case, there is one qualifier: the `and` method combination type (it can be retrieved by the function `method-qualifiers`), and the specializers are twice the class of the objects.

Once we have made sure this method exists (`find-method` would trigger an error otherwise), we must perform the same check on the whole super-hierarchy (the topmost, standard class excepted). As its name suggests, the function `class-direct-superclasses` returns a list of direct superclasses for some class. We can then recursively call our test function on this list.

By hooking the code excerpts from listings 13 and 14 into the skeleton of listing 12, we have completed our check for the “binary completeness” property.

4 Conclusion

In this paper, we have described why binary methods are a problematic concept in traditional object-oriented languages: the relationship between types and classes in the context of inheritance, and the need for privileged access to the internal representation of objects make it difficult to implement.

From the CLOS perspective, we have demonstrated that implementing binary methods is a straightforward process, for at least the following two reasons.

1. The covariance / contravariance problem does not exist, because CLOS generic functions natively support multiple dispatch.
2. When privileged access to internal information is needed, the dynamic nature of Common Lisp provides solutions that are unavailable in statically typed languages. Besides, the package system is completely orthogonal to the object-oriented layer and is pretty liberal in what you can access and how (admittedly, at the expense of breaking modularity just as in other languages).

From the MOP perspective, it is also important to realize that we have not just made the concept of binary methods accessible; we have implemented it *directly* and *explicitly*: we have shown ways to not only implement it, but also enforce a correct *usage* of it, and even a correct *implementation* of it. To this aim, we have actually programmed a new object system which behaves quite differently from the default CLOS. CLOS, along with its MOP, is not only an object system. It is an object system designed to let you program your own object systems.

References

- [Bruce et al., 1995] Bruce, K. B., Cardelli, L., Castagna, G., Eifrig, J., Smith, S. F., Trifonov, V., Leavens, G. T., and Pierce, B. C. (1995). On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242.
- [Castagna, 1995] Castagna, G. (1995). Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447.
- [Keene, 1989] Keene, S. E. (1989). *Object-Oriented Programming in Common Lisp: a Programmer’s Guide to CLOS*. Addison-Wesley.
- [Kiczales et al., 1991] Kiczales, G. J., des Rivières, J., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA.
- [Paepcke, 1993] Paepcke, A. (1993). User-level language crafting – introducing the CLOS metaobject protocol. In Paepcke, A., editor, *Object-Oriented Programming: The CLOS Perspective*, chapter 3, pages 65–99. MIT Press. Downloadable version at <http://infolab.stanford.edu/~paepcke/shared-documents/mopintro.ps>.

Work-in-Progress Track

OpenMusic: Design and Implementation Aspects of a Visual Programming Language

Carlos Agon, Jean Bresson, Gérard Assayag
IRCAM - CNRS UMR STMS
Music Representations Research Group

1 Introduction

OpenMusic (OM) is a computer-aided composition environment developed at Ircam since the end of the 90s [1] [6] [7]. It is a complete functional programming language extending Common Lisp with a visual specification.

Thanks to graphical tools and protocols, the user/programmer can create functions and programs using arithmetic or logic operations, and make use of other programming concepts like functional abstraction and application, iteration or recursion. As we shall demonstrate in this article, he/she can also benefit from the powerful object protocol of CLOS (Common Lisp Object System [11]).

The musical issues and compositional relevance of this environment, widely discussed in various related publications, are voluntarily left aside; in the present paper we shall rather focus on different aspects of the programming language design and features.

2 Language Architecture

The elements of the visual language can be divided in two categories: the *meta-objects* are the “traditional” language primitives (functions, programs, classes, instances, types, etc.) and the *visual components* (or *visual meta-objects*) constitute the visual part of the language and provide its graphical representation and user interactions.

2.1 Meta-Objects

In CLOS, the classes, generic functions, methods, and other element of the language are meta-object classes, instances of the class *standard-class* [12]. These classes (respectively *standard-class*, *standard-generic-function*, *standard-method*, etc.) can therefore be subclassed and extended by new classes. This what is systematically done in OM for defining the language meta-objects (e.g. *OMClass*, *OMGenericFunction*, *OMMethod*, etc.) *OMClass*, for instance, is defined as a subclass of *standard-class* as follows:

```
(defclass omclass (standard-class) ())

> #<standard-class omclass>
```

In order to make new classes instances of *OMClass* instead of *standard-class*, one can then use the *:metaclass* initarg in class definition as follows:

```
(defclass my-class ()
  ((slot1 :initarg :slot1 :accessor slot1)
   (slot2 :initarg :slot2 :accessor slot2))
  (:metaclass omclass))

> #<omclass my-class>
```

That way it is possible to extend *standard-class* in order to set particular behaviours and properties related to specific aspects of the visual language (icons specification, documentation, persistence, behaviours in the graphical user interface, etc.) Every class defined as an *OMClass* instead of *standard-class* will therefore possibly be handled in the visual part of the language.

Specific protocols are established for the creation of OM meta-objects. The macros *defclass!* and *defmethod!* expand as calls to *defclass* et *defmethod* that specify the appropriate metaclass and allow for the setting of the particular attributes of the corresponding meta-object. For instance, *OMClass* has a slot called *icon* containing an icon ID converted as a picture icon when it is represented in the visual language. An *OMClass* can therefore be created directly as follows:

```
(defclass! my-class2 () ()
  (:icon 21))

> #<omclass my-class2>
```

The same principle applies for generic functions and methods. In the following example, the keywords *icon*, *initvals* and *indocs* allow to specify the attributes of the *OMMethod* instance which is created, that will be used to determine respectively the icon for the graphical representation of this methods, the default values, and a documentation for each of its arguments:

```
(defmethod! my-method (arg1 arg2)
  :icon 123
  :initvals '(0 0)
  :indocs '("argument1" "argument2")
  (+ arg1 arg2))

> #<ommethod my-method>
```

2.2 Visual Components

The visual aspects of the language principally manifest themselves through the *boxes* (class *OMBox*) and the *editors* (class *Editor*). These are however still “non-graphic” objects in the environment: a box is a “visual meta-object”, i.e.

a syntactic specification of the visual language. Visual components generally refer to other elements of the language (e.g. functions, classes or programs) and represent them in the user interface. The actual graphic level (i.e. the GUI elements) encapsulate these non-graphic visual meta-objects via the class *OMFrame* and its subclasses (*BoxFrame*, *InputFrame*, *EditorFrame*, etc.)

The visual aspects are thus not simple interfaces on the language but a set of meta-object properties and graphical components taking part in the language specification. Figure 1 shows the class architecture corresponding to the main elements of the language.

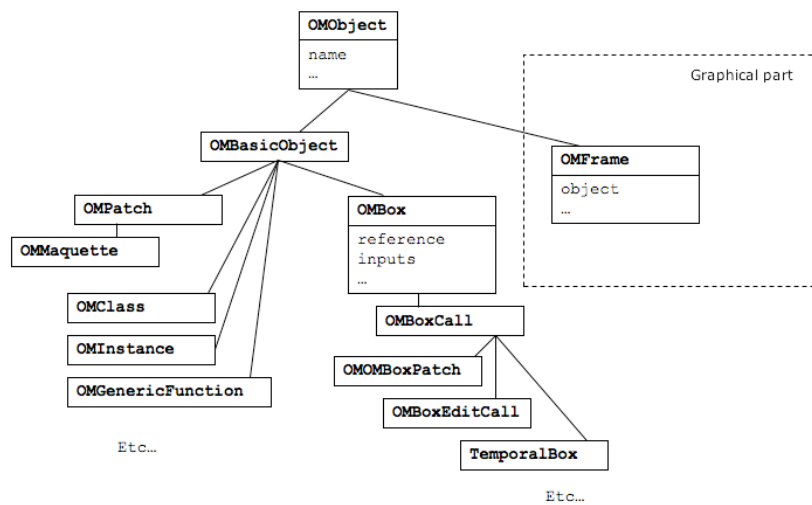


Figure 1: Simplified architecture of the OM visual programming language.

3 Visual Programming

Two main visual components were cited above: boxes and editors. The boxes are represented by frames surrounded by inputs and outputs, according to the object they refer to. They are possibly connected with one another within a visual program via these inputs and outputs. Most of the objects are also associated to an editor, which allow for their “manual” building and edition.

Patches represent visual programs and are the main entry-point in the OM programming framework (class *OMPatch* in Figure 1). They are associated to an editor in which the user/programmer creates and assembles functional units represented by boxes.

Figure 2 shows a patch implementing simple arithmetic operations.

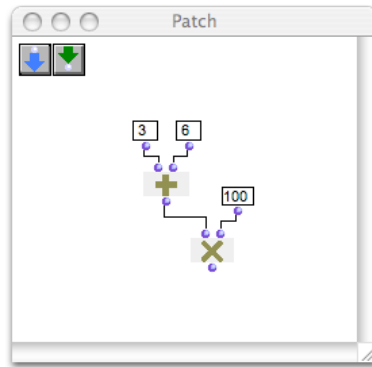


Figure 2: A patch implementing the operation $(3 + 6) \times 100$.

Each box refers to a functional object: in the example of Figure 2, the boxes refer to functions (+ and \times) or to constant values (3, 6 and 100). The functions can be classical Common Lisp functions (*standard-function*, *standard-method*, etc.) or OM functions (*OMMethod*). They can be built-in functions, included in the Lisp image, or user defined functions created or loaded dynamically while using OM.

As mentioned above, a box has a variable number of inlets and outlets so that it can be connected to other boxes. Inlets are visible at the top of the boxes, and outlets at the bottom. A set of connected boxes therefore constitutes an acyclic graph that corresponds to a functional expression. The patch in Figure 2 corresponds to the following Lisp expression:

```
(* (+ 3 6) 100)
```

The graph defined in a patch can be evaluated at any point, as a Lisp expression does. The evaluation of a box, triggered by a user action, is a call to the referred function. The arguments of this function call are the results of the evaluation of the boxes connected to the various inlets of this box. A recursive sequence of evaluations therefore occurs in order to reduce the Lisp expression according to the functional composition defined by the connections, which corresponds to the execution of the program.

The evaluation of the box \times in Figure 2 starts this reduction process: the result of box 100 (i.e. the value itself) is multiplied to that of box +, and so forth. Then the final result is returned:

```
> 900
```

The function responsible for the evaluation of the boxes is the method *omng-box-value*:

```

; Eval the output indexed by 'numout' for the box 'self'
(defmethod omNG-box-value ((self OMBoxCall) &optional (numout 0))
  (let ((args (mapcar #'(lambda (input)
                        (omNG-box-value input))
                      (inputs self))))
    (nth numout (multiple-value-list
                 (apply (reference self) args)))))
)

```

This method is specialized for the different types of boxes (subclasses of *OMBox*). Here, *OMBoxCall* is a box that refers to a function. Note that *omng-box-value* called on a box input reports the call on the box connected to this input, following the links established by the connections in the patch.

3.1 Functional Abstraction

Functional abstraction basically consists in making some elements of a program become variables. Inputs and outputs, also represented by boxes, can be introduced in an OM patch. They will represent these possible variables in the program defined in this patch.

Starting from the example in Figure 2, it is possible to create the function $f(x, y) = (x + 6) \times y$ by adding two inputs and an output connected to the program as shows *patch1* in Figure 3 (a).

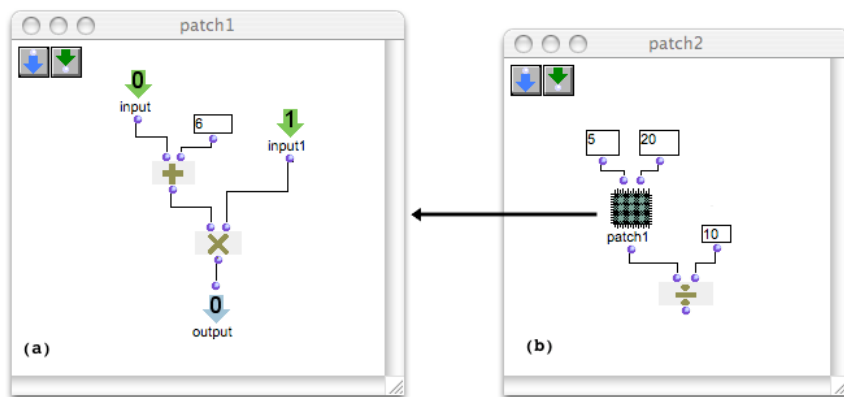


Figure 3: (a) Definition et (b) application of a function. Abstraction is carried out by making variable some elements of the program.

Then *patch1* now corresponds to a function definition. The corresponding Lisp expression would be:

```
(lambda (x y) (* (+ x 6) y))
```

As a function definition, this expression can also be expressed as follows:

```
(defun myfunction (x y) (* (+ x 6) y))
```

Patch1 can then be used in another patch, and is then considered as a function with 2 arguments and 1 output value, as in *patch2* on Figure 3 (b). In this patch can be set the values of the abstraction variables (functional application).

In this example, we see a new type of box, which refers to the patch (the box labelled *patch1*). Its evaluation corresponds to the Lisp call:

```
(myfunction 5 20)
```

The new program (*patch2*) therefore corresponds to the expression:

```
(/ (myfunction 5 20) 10)
```

In these abstraction/application mechanisms, the patch is converted into a Lisp function. A function called *compile-patch* carries out this conversion by a recursive call to a code-generating method (called *gen-code*) on the functional graph that constitutes the patch. During this recursive call to *gen-code*, each box generates the Lisp code corresponding to its referring object. The newly generated Lisp expression is then compiled and the resulting function is attached to the patch (the class *OMPatch* has a dedicated slot called *code*).

The evaluation of a box referring to this patch thus consists in the application of the values connected to its inputs to the compiled function:

```
; Eval the output indexed by 'numout' for the box 'self'
(defmethod omNG-box-value ((self OMBBoxPatch) &optional (numout 0))
  (let ((args (mapcar #'(lambda (input)
                        (omNG-box-value input))
                    (inputs self))))
    (unless (compiled? (reference self))
      (compile-patch (reference self)))
    (nth numout (multiple-value-list
                 (apply (code (reference self)) args)))
  ))
```

Within a patch editor, the program can thus be modified and partially executed. From the outside, however, it is a box corresponding to an abstract function. This function can therefore be used later on in different contexts and purposes. The multiple occurrences of a patch box in other patches will all refer to the same function.

Abstractions can also be used in their own definitions, hence implementing the notion of recursion. Figure 4 shows a patch corresponding to the recursive function “factorial”:

```
(defun factorial (x)
  (if (= x 0) 1
      (* x (factorial (- x 1))))
  ))
```

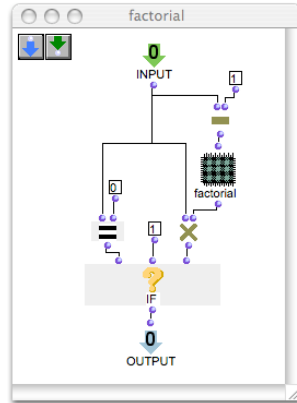



Figure 4: The recursive factorial function in OM.

3.2 *Lambda* Functions

In functional languages data and functions are equally considered as “first-class citizens”. A Lisp function can thus be considered as data and inspected or constructed in the calculus. This allows for the creation of “higher-level functions”, i.e. functions that take other functions as arguments, or producing functions as output values.

OM boxes can be set to a “lambda” state so as to return not the result of its reference’s functional application, but its reference as a function object. When a patch box is in mode “lambda”, a small *lambda* icon is displayed on it. A box like *patch1* in Figure 3 (b), for example, if it is set to this lambda mode, will not return a value anymore (22, in this example), but the functional definition (`(lambda (x y) (* (+ x 6) y))`) which will be used and eventually called as such in the continuation of the program execution (see Figure 5).

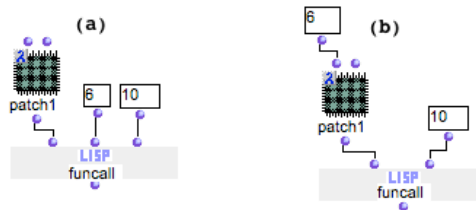


Figure 5: (a) Creation of a lambda form in a visual program. The patch box *patch1* is in mode “lambda” and returns a function, called using the *funcall* box and arguments 6 and 10. (b) Curryfication: the previous function is converted to a function of one single argument by explicitly setting one of the input values in the lambda form.

The curryfication (i.e. transformation of a function of n arguments into a function of $n - 1$ arguments) can also be carried out using the lambda mode, by connecting explicitly one value to some inputs of the patch box as shown in Figure 5 (b).

3.3 Local Fonctions

Complementarily to the abstraction mechanism detailed above, it is possible to create sub-programs (or sub-patches) which actually are local functions, defined only within the local context of a patch. These sub-patches are graphically differentiated with the color of their referring boxes.

For instance in the example of Figure 3 *myfunction* is defined in the environment, which would correspond to the following expressions:

```
; patch1
(defun myfunction (x) (* (+ x 6) 100))

; patch2
(defun myprogram (x) (/ (myfunction x) 10))
```

with a local function, we have the possibility to obtain something similar to:

```
(defun myprogram (x)
  (flet ((myfunction (x) (* (+ x 6) 100)))
    (/ (funcall myfunction x) 10)))
```

In this case, *myfunction* does not exist outside *myprogram*. As a consequence while all boxes referring to an abstraction point to a unique patch, local functions can be duplicated and edited independently in each of their occurrences. It is also possible to detach patches from their abstraction (by creating a local copy of the function) or conversely to define a global abstraction from a local function.

3.4 Local Variables

In order to simulate the Lisp *let* statement and to allow for the creation of local variables, it is possible to set the function call boxes to a third state called *ev-once* (see Figure 6). In this mode the box is evaluated only once during an evaluation process.

The example in Figure 6 shows a patch in which the topmost box $+$ is connected to various other boxes, so that in principle it should be called various times (and recursively call each time the possible boxes it depends on). The corresponding expression for this example would be:

```
(list (+ 7 2) (+ (+ 7 2) 3))
```

As the box $+$ in mode *ev-once* (see little icon at the top-left of the box), the result of the call $(+ 7 2)$ box will be stored after its first call, so that this example actually corresponds to :

```
(let ((var (+ 7 2))) (list var (+ var 3)))
```

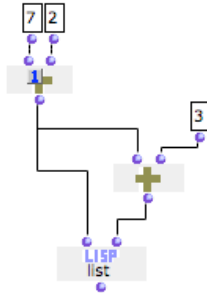


Figure 6: Creation of local variable. The box in *ev-once* mode (at the top) is evaluated only once during a global eval process.

This case is a simple example, but the factorization of the call $(+ 7 2)$ in a local variable can be crucial in more complex processes. In case of non-deterministic processes (e.g. involving a random call), the *ev-once* state will also ensure that the box provides the same results to its different callers during a same evaluation context.

It is also possible to completely lock a box so that it computes and stores its result once and keep it for all the following evaluations until the box is unlocked. This would rather correspond to a global variable.

3.5 Control Structures

Various control structures commonly used in programming languages (conditionals, iterations, etc.) are available in the OM visual programming framework.

Figure 7 shows an example of an *omloop*, which represents an iterative process (the *loop* Lisp macro). The *omloop* box visible on the left is associated to a special patch editor allowing one to define the behaviour of the program during this iteration.

In this example the iteration is done via the *list-loop* iterator (other available iterators include *while*, *for*, *on-list*, etc.), on a list given as the input of the loop. At each step of the iteration, hence for each element in the list, another control structure is used: conditional structure *omif*, which corresponds to the Lisp *if* statement (also present in the previous example of Figure 4). Here, the values from the list are incremented if they are inferior to a given threshold. The successive results are collected in a new list which is returned as the result of the iteration. This loop thus corresponds to the following Lisp expression:

```
(lambda (list)
  (loop for x in list
        collect (if (>= x 5) x (+ x 5))))
```

The *file-box* tool is another example of a visual iteration, performing the equivalent of an *omloop* within a *with-open-file* statement, i.e. with an input and/or output access to a file stream (see Figure 8).

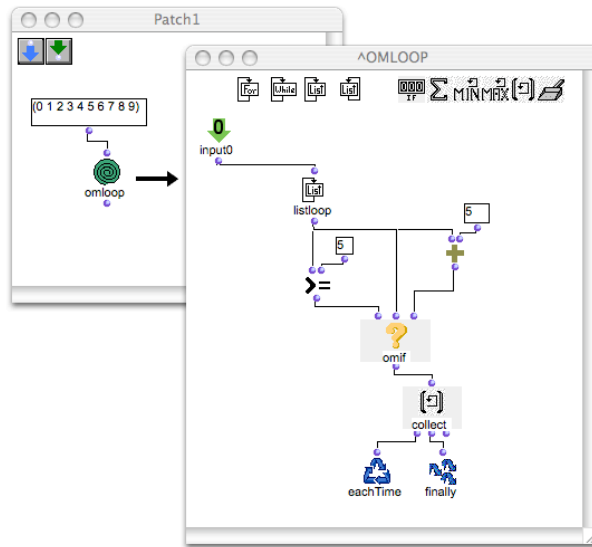


Figure 7: *omloop*: iterative process.

The patch in Figure 8 corresponds to the expression:

```
(lambda (path list)
  (with-open-file (s path)
    (loop for item in list do (write-line item s))))
```

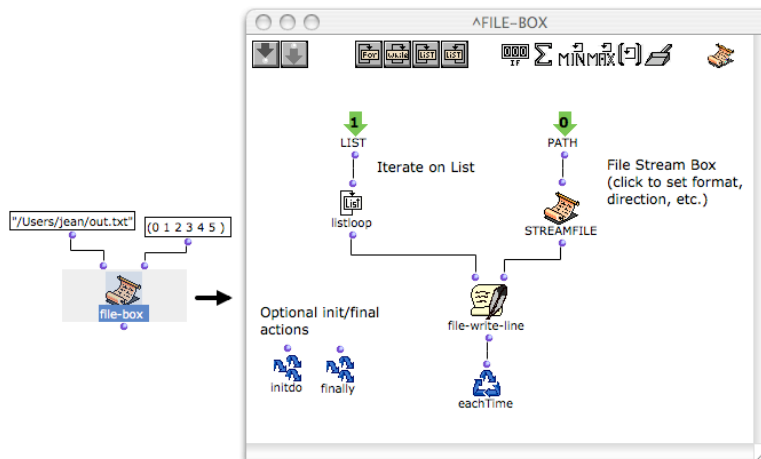


Figure 8: *filebox*: i/o access on file streams. The *streamfile* box represents the stream declaration, initialized with a pathname.

4 Object-Oriented Programming

In addition to the functional programming features presented above, OM also offers object-oriented programming facilities.

The main use that is actually done by composers of the object-oriented programming is generally to create instances of in-built classes and methods. OM includes some predefined musical or general-purpose classes and functions organized in a hierarchical package architecture. Figure 9 shows the OM packages browser window.

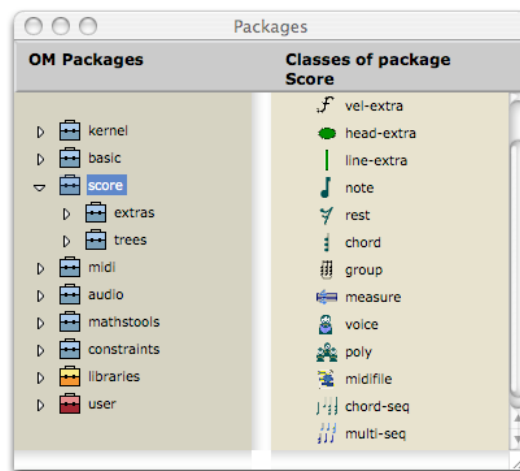


Figure 9: OM *packages* window.

As we shall demonstrate forthwith, however, users can also define their own classes and methods in the visual language.

They are also provided with means to get further in object-oriented programming with meta-object programming tools. Indeed, as mentioned in the first part of the paper, the meta-object protocol provides reflexive properties, so that the elements that constitute the language can become the objects of processing of this same language. The meta-objects and visual meta-objects classes that constitute a program (classes, functions, methods, boxes), as well as their corresponding behaviours, can therefore be defined and modified dynamically while running this program (see [3] for a detailed description of meta-object programming in OM).

4.1 Classes

Figure 10 shows the class tree of one of the OM subpackages: the *score* package. The class tree editor is accessible via the corresponding icon on the package browser window. It shows the different classes defined in this package and their possible inheritance relationships.

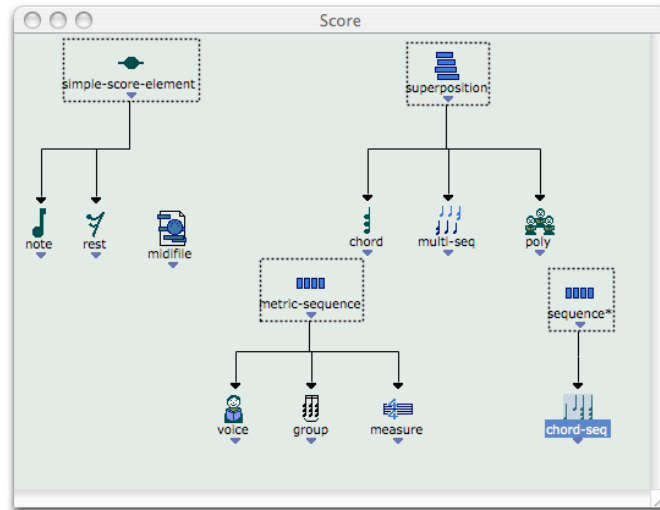


Figure 10: Class tree editor of package *score*. The dotted frames indicate *alias* boxes that refer to classes from other packages.

Users can create new classes in the *user* package: inheritance relationships can be dynamically created and edited as well by setting/modifying the arrow-shaped graphical connections between user class boxes and/or OM predefined class boxes (see Figure 11).

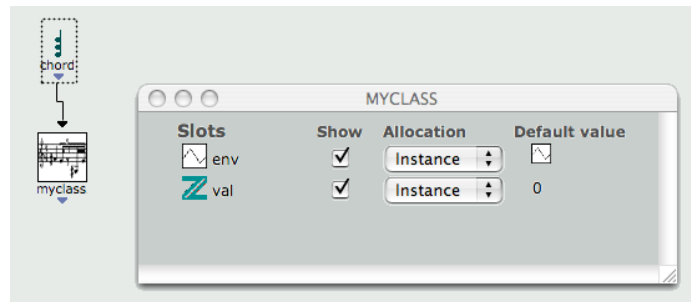


Figure 11: Creation of a user class *myclass*, extending the OM class *chord* (in this example, *chord* is an alias, since the *chord* class is not in the *user* package). The class editor is open at the right of the figure: two additional slots are created.

Figure 11 also shows the editor for the newly created class. Slots can be added to the class, which types and initialisation values are set graphically in this editor. The equivalent Lisp expression, automatically generated and evaluated in this situation, is:

```

(defclass! myclass (chord)
  ((env :accessor env :initarg :env
        :allocation :instance
        :initform (make-instance 'bpf))
   (val :accessor val :initarg :val
        :allocation :instance
        :initform 0))
  (:icon 212)
  (:documentation ""))

```

4.2 Instances and Factories

In order to use classes in visual programs, a special kind of box is used: the *factory* box. A factory is a box attached to a given class, which represents a functional call generating instances of this class (*make-instance*) and allows to set/get the values of the different slots of this instance. At the top of Figure 12 is visible the *note* class factory. The various inlets/outlets of the factory box represent set/get accesses on the instance itself (first in/outlet from left) and to the different slots of the class (for example, pitch, velocity, duration, etc. for the class *note*).

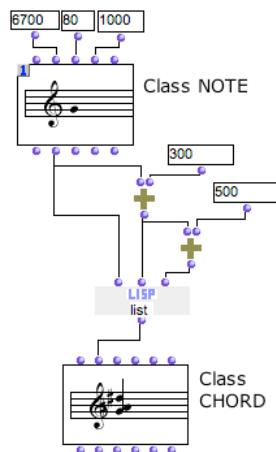


Figure 12: The use of factory boxes in OM. A *note* object is instantiated from integer values corresponding to its pitch, velocity and duration. The pitch is used and processed in order to create a list of three values, in turn used to instantiate a *chord* object.

Figure 12 corresponds to the following expression:

```

(let ((note (make-instance 'note :pitch 6700
                          :vel 80
                          :dur 1000)))
  (make-instance 'chord :pitches (list (pitch note)
                                       (+ (pitch note) 300)
                                       (+ (+ (pitch note) 300) 500))))

```

The predefined musical object classes provided in OM have dedicated editors (e.g. score editors) associated to the corresponding factory boxes, and which allow to edit or just visualize the current value contained in these boxes (i.e. the last created instance). The *factories* therefore make it possible to generate and/or store the state of a data set or structure at a given moment in the calculus (i.e. at a given position in the graph defined in the patch), and at the same time provide a direct access to these data via the editor [2]. That way, they constitute privileged entry-points for the introduction of data and the interaction of the user/programmer with the program [5].

4.3 Generic Functions, Methods

The polymorphism of the generic functions in CLOS is also integrated in the OM visual programming features. New generic functions can be created graphically, as well as their different methods specializing the different possible types of their arguments. It is also possible to add new methods to existing generic functions in order to specialize them for specific types.

Figure 13 shows the editor for the generic function *om+*, which lists its four existing methods. The editor of a new method being defined is also open. It allows to define a visual program corresponding to the method process, and is similar to a patch editor (except for the input types management and the possible *:before*, *:around* and *:after* statements).

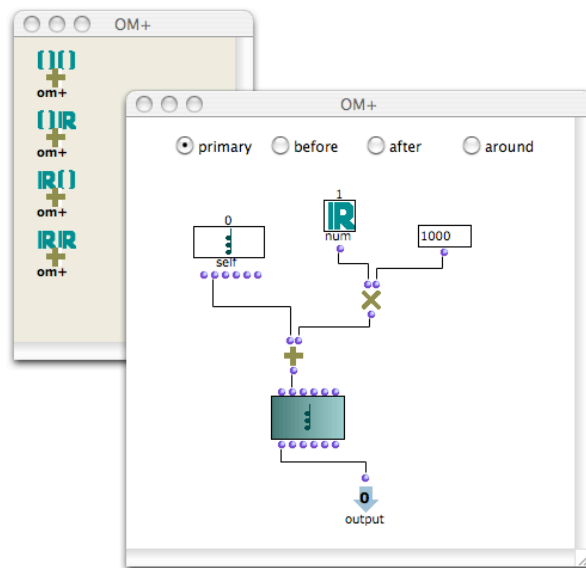


Figure 13: Method definition. A new method specializes the generic function *om+* for arguments of types *chord* and *number*

The method created in Figure 13 corresponds to the following Lisp definition:

```
(defmethod! om+ ((c chord) (n number))
  (make-instance 'chord
    :pitches (om+ (pitches chord) (om* n 1000)))
)
```

Among the different other object-oriented programming features available in OM, are also the possibility to define a specific processing function called at creating an instance of a class, or to redefine graphically the accessor methods of its different slots.

5 Persistence

The main window of the OM environment is called a *workspace* and is similar to a classical OS desktop. In this workspace the user creates programs (patches) and organizes them in a directory tree. Figure 14 shows an OM workspace. Each icon represents a patch or a folder containing patches or sub-folders.

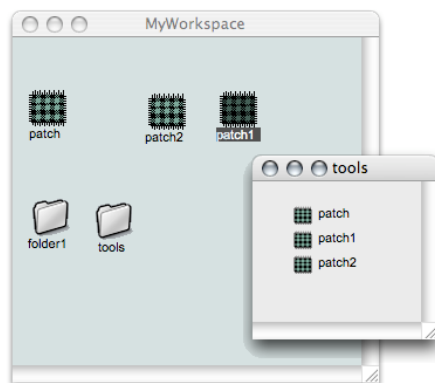


Figure 14: A *workspace* in OM.

This organisation reflects a real file/folder architecture on the disk: patches are “persistent” objects. They are saved as Lisp files which evaluation recreate the original visual programs.

Similarly, user-defined meta-objects (classes or methods), organized in packages and accessible via the package tree window (Figure 9), are also saved as Lisp forms in files on the disk. That way, the user can save his workspace’s contents and later reload his programs, classes and functions as in a traditional programming environment.

6 Current Implementation Issues

OM is one the successors of the PatchWork visual programming environment [14]. It has been initially developed for Macintosh computers using MCL Lisp compiler. In 2005, with version OM 5, the code was refuted so as to improve modularity and reduce Lisp and/or platform dependencies of the environment [8]. An API has been specified, gathering graphical features, user interfaces and non-ANSI CL parts of the code, in order to facilitate portability on new Lisp compilers and platforms. This API has been implemented on MCL and on Allegro CL for Windows. A Linux implementation using SBCL and GTK+ graphical toolkit is also currently in progress. The programming protocol defined by the OM API then allows to systematically interpret the OM code according to the targeted platforms.

Since MCL was not ported on the new Macintosh computers / Intel x86, the need for a new reliable, efficient Lisp with GUI creation toolkit led us to start a new port of OM on LispWorks, which will probably be used as a common support for Mac Intel, Mac PPC and Windows versions of OM. OM 6 / LispWorks for Mac has been distributed as a beta test version on February 2008.

7 Conclusion

We presented some aspects about the OpenMusic visual programming language, particularly concerning functional and object-oriented programming features. Many works have also been carried out in OM regarding constraint programming: various constraints solver are integrated in it, and were used in a large number of musical applications (see [15], [13], [10]).

Complementarily to these programming tools, OM provides an important library of classes, data structures and predefined functions allowing to head programming toward musical and compositional applications. The more general-purpose tools are integrated in the OM image, while more specific or aesthetically oriented ones are dynamically loaded via external user libraries.

Many musical works have been created with OM during the past 10 years. *The OM Composer's Books* provide varied interesting examples of these applications of visual programming for music composition [4] [9].

References

- [1] Agon, C. *OpenMusic : Un langage visuel pour la composition musicale assistée par ordinateur*. PhD Thesis, Université Pierre et Marie Curie (Paris 6), 1998.
- [2] Agon, C. and Assayag, G. "Programmation visuelle et éditeurs musicaux pour la composition assistée par ordinateur", *14ème Conférence Francophone sur l'Interaction Homme-Machine IHM'02*, Poitiers, France, 2002.

- [3] Agon, C. and Assayag, G. “OM: A Graphical extension of CLOS using the MOP”, *Proceedings of ICL'03*, New York, USA, 2003.
- [4] Agon, C., Bresson, J. and Assayag, G. (Eds.) *The OM Composer's Book*, Vol. 1, IRCAM – Editions Delatour France, 2006.
- [5] Assayag, G. and Agon, C. “OpenMusic Architecture”, *Proceedings of the International Computer Music Conference*, Hong Kong, 1996.
- [6] Assayag, G., Agon, C., Fineberg, J. and Hanappe, P. “An Object Oriented Visual Environment for Musical Composition”, *Proceedings of the International Computer Music Conference*, Thessaloniki, Greece, 1997.
- [7] Assayag, G., Rueda, C., Laurson, M., Agon, C. and Delerue, O. “Computer Assisted Composition at IRCAM: From PatchWork to OpenMusic”, *Computer Music Journal*, 23(3), 1999.
- [8] Bresson, J., Agon, C. and Assayag, G. “OpenMusic 5: A Cross-Platform release of the Computer-Assisted Composition Environment”, *Proceedings of the 10th Brazilian Symposium on Computer Music*, Belo Horizonte, MG, Brasil, 2005.
- [9] Bresson, J., Agon, C. and Assayag, G. (Eds.) *The OM Composer's Book*, Vol. 2, IRCAM – Editions Delatour France, 2008.
- [10] Bonnet, A and Rueda, C. “Situation: Un langage visuel basé sur les contraintes pour la composition musicale”, in *Recherches et applications en informatique musicale*, Chemillier M. and Pachet, F. (Eds.), Hermes, 1998.
- [11] Gabriel, R. P., White, J. L. and Bobrow, D. G. “CLOS: Integration Object-oriented and Functional Programming”, *Communications of the ACM*, 34(9), 1991.
- [12] Kiczales, G., des Rivières, J. and Bobrow, D. G. *The Art of the Metaobject Protocol*, MIT Press, 1991.
- [13] Laurson, M. “PWConstraints”, *Symposium: Composition, Modélisation et Ordinateur*, IRCAM, Paris, 1996.
- [14] Laurson, M. and Duthen, J. “Patchwork, a Graphic Language in PreForm”, *Proceedings of the International Computer Music Conference*, Ohio State University, USA, 1989.
- [15] Siskind, J. M. and McAllester, D. A. “Nondeterministic Lisp as a Substrate for Constraint Logic Programming”, *Proceedings of the 11th National Conference on Artificial Intelligence*, AAAI Press, 1993.

CLAZY: Lazy Calling for Common Lisp

Marco Antoniotti

Dipartimento di Informatica, Sistemistica e Comunicazione,

Università Milano Bicocca

U14 - Viale Sarca 336, I-20126 Milan, ITALY

Abstract

This document contains a description of a Common Lisp extension that allows a programmer to write functional programs that use *normal order* evaluation, as in *non-strict* languages like Haskell. The extension is relatively straightforward, and it appears to be the first one such that is integrated in the overall Common Lisp framework.

1 Introduction

Common Lisp is a functional language (and also an imperative, object-oriented one, which, moreover, can be used in a declarative fashion). As a functional language it falls in the category of *strict* languages like ML and OCaml, unlike Haskell, which is in the category of *normal-order* or *lazy* languages.

That is to say that the following code will enter an infinite loop, should it be executed at the Common Lisp prompt.

```
cl-prompt> (defun si (condicio ergo alternatio)
             (if condicio
                 ergo
                 alternatio))
```

SI

```
cl-prompt> (si t 42 (loop))
```

In a *lazy* language the function `si` (if in Latin) would return 42 instead of waiting for the form `(loop)` to produce a value.

In a bout of Haskell envy, I decided to look into some extensions to Common Lisp that would introduce ways to program in a lazy way. The result may sound *crazy*, and, in fact, a little bit it is.

The notion of *lazy evaluation* dates back to the Algol days and the notion of *by-name* parameter passing. In the Lisp camp, the best known way to introduce a form of lazy evaluation is to implement *streams* as described in *Structure and Interpretation of Computer Programs* (SICP) [1]; incidentally this form of

lazy evaluation is also used by Okasaki [5] in his exposition of *functional data structures* in ML.

In SICP, streams are implemented using two primitives, `force` and `delay`, which can then be used to build a lazy container (the “stream”) using a *macro* `cons-stream`, and two accessors `head` and `tail`. A sufficient implementation in Common Lisp is the following:

```
(defmacro delay (expr) `(lambda () ,expr))

(defun force (thunk) (funcall thunk))

(defmacro cons-stream (head tail) `(cons ,head (delay ,tail)))

(defun head (lazy-stream) (car lazy-stream))

(defun tail (lazy-stream) (force (cdr lazy-stream)))
```

At this point there are several Common Lisp packages floating around the net, that implement this flavor of lazy evaluation. E.g., `Heresy` [4], `funds` [2] and `FSet` [3] are exemplars of this approach. `CLAZY` goes off a (different) tangent and provides a more fundamental way to build such lazy constructions.

1.1 Limits of the delay/force Duo

Given `delay` and `force`, one could always implement the operator `si` as a macro using `delay`, as in

```
(defmacro si (condicio ergo alternatio)
  `(if (force ,condicio)
      (force (delay ,ergo))
      (force (delay ,alternatio))))
```

but this is a bit unsatisfactory as far as Haskell `envy` is concerned. `si` cannot be `funcalled` in any meaningful way and cannot be passed around as we would expect a regular function to be. A different solution is needed.

2 Defining and Calling Lazy Functions

It is possible to come up with a more satisfactory solution that will allow us to bypass `delay` and `force`, at the price of tweaking the “calling convention”. Then we can write `si` as:

```
(deflazy (condicio ergo alternatio)
  (if condicio ergo alternatio))
```

where `deflazy` defines both *lazy* and *strict* versions of the operator.

The *lazy function* `si` can now be called as

```
CL prompt> (lazy:call #'si t 42 (loop))
42
```

I.e., `lazy:call` is the lazy version of `funcall`. The complexity of writing lazy code is thus moved to the call points. This may or may not be desirable, but it can be argued that this is a slightly better way than having to manually **force** expressions. In any case, the CLAZY approach still uses the `delay/force` duo under the hood, and they are available for more manual intervention.

From the example above, it should be apparent that `lazy:call` is a macro that does something special with the call, recognizing functions that are defined via `deflazy`. As a matter of facts, the expansion of `lazy:call` looks like this:

```
(lazy:call <op> <arg1> <arg2> ... <argN>)
⇒
(funcall <lazyfied op>
        <thunked arg1>
        <thunked arg2>
        ...
        <thunked argN>)
```

The “lazy” version of `<op>` is defined by `deflazy` and each `<thunked argi>` is a closed over version of the argument as if `delay` was invoked on it.

Of course, a simple version of such idea can be easily implemented with a few macros, however, a well integrated version within the overall Common Lisp environment requires a few more bits and pieces. As example, CLAZY wants to make the analogy between `lazy:call` and `funcall` as tight as possible. This means that we need a way to pass (almost) regular `lambda`'s to `lazy:call`. This can be done the special operator `lazy`, which acts as `function`; moreover, it does wrap around the `function` operator as expected. See Figure (1) for an example.

Extra work is needed to handle `&optional` and `&key` parameters, but the overall design lies in this tweaking of the calling point and in allowing lazy functional objects to be passed around as regular functions (of course to be called via `lazy:call`).

2.1 Example: Lazy Functional Conses

Another example which turns out to be more easily realizable with CLAZY is the standard “*conses are functions*” one.

```
CL prompt> (lazy:call (lazy #'(lambda (condicio ergo alternatio)
                             (if condicio
                                 ergo
                                 alternatio)))
              t
              (+ 20 20 2)
              (loop))
```

42

Figure 1: An example of the use of the special operator `lazy`.

```
(deflazy consing (head tail)
  (lambda (selector)
    (ecase selector
      (car head)
      (cdr tail))))

(deflazy head (cons)
  (funcall cons 'car))

(deflazy tail (cons)
  (funcall cons 'cdr))
```

Now, we can build truly lazy lists¹

```
CL prompt> (defparameter ll
            (lazy:call 'consing
                      1
                      (lazy:call 'consing
                                  (loop)
                                  (lazy:call 'consing
                                              3
                                              (loop)))))

LL

CL prompt> (head (tail (tail ll)))
3
```

Or the usual streams from SICP as the integers here below.

¹Note where the `(loop)` calls appear.


```
(defun integers-from (n)
  (lazy:call 'consing n (integers-from (1+ n))))

(defparameter integers (integers-from 0))
```

Yet, it must be noted that having normal order evaluation at one's disposal naturally leads to the implementation of much more complex and sophisticated functional software, as in the case of the integrators in Section 3.5 of [1].

2.2 Extra Considerations

CLAZY is supposed to be used in a very controlled way. While it is true that it adds *normal order evaluation* to Common Lisp, the user must remember that s/he is not using Haskell or a similar language. At its core, Common Lisp is a *strict* language, which allows side-effects; not a good mix to produce lazy code in a careless way. See also the note on *normal order evaluation* in Section 3.5 on streams of [1].

3 Reference Implementation

The CLAZY reference implementation can be found at common-lisp.net. The implementation lies within a package nicknamed LAZY and is based on the macros `lazy:call`, `lazy:deflazy`, and `lazy:lazy`.

The `lazy:call` macro is used at calling time (as the name implies). The `deflazy` macro is used to define functions. The `lazy` “special operator” returns a functional object that should be called in a lazy way, although the system is set up in such a way to “pass through” constant values (as tested by `constanp`).

The reference implementation is based on the pre-processing of lambda list arguments by `deflazy`: each argument is substituted by an internal name, which is expected to be bound to a *think* generated by `lazy:call` as per `delay`. In the body of a lazy function (or of a *lazy lambda*) each lambda list argument is actually re-defined as a `symbol-macrolet`, which expands in the appropriate `force` call. `deflazy` installs the lazy version of the function being defined in the property list of the function name.

Ordinary Lambda List Processing. As noted before, CLAZY pre-processes `&optional` and `&key` arguments in such a way to preserve the expected Common Lisp semantics. E.g., the calls in Figure (2) yield 42 as expected. On the contrary, the implementation does not treat `&rest` arguments in a special way (i.e., they are not *thunked*), this is because there is no way to access the list forming machinery in Common Lisp when `&rest` arguments are present; in a lazy piece of code, the list in the `&rest` argument will contain the actual thunks generated as if by `delay`.

```

(lazy:call (lazy (lambda (x &key (y (loop) y-supplied-p))
                (if y-supplied-p y (+ x 21))))
          21)

(lazy:call (lazy (lambda (x &key (:y yy) (loop))
                (if x (+ x 21) yy)))
          21)

(lazy:call (lazy (lambda (x &key (:y yy) (loop))
                (if x (+ x 21) yy)))
          nil :y 42))

```

Figure 2: `&key` arguments are dealt with as expected. The answer is always, as expected, 42.

4 Conclusions

CLAZY is an exercise in Common Lisp style, which is also useful. The CLAZY library shows how, at the price of introducing a special call operator (`lazy:call`), it is possible to introduce *normal order* or *lazy* evaluation in Common Lisp. The extension has the following desirable characteristics: (i) it does not require the construction of a full blown interpreter implementing lazy evaluation, and (ii) thanks to the `deflazy` macro it allows a programmer to write code in the most natural way. It is much more difficult to achieve the same effect in any other language than Common Lisp, even when the language has macros. It is the interaction of macros and `symbol-macrolet` that makes CLAZY possible.

Of course, once this basic machinery is in place, extra Common Lisp incantations can be made and reader macros put in place as desired.

CLAZY is not perfect of course. The main open issue to complete the integration within the frame provided by Common Lisp is to work out a way to deal with CLOS methods. One way to achieve this would be to automatically define a method specializing on thunks for a given generic function. While this may work, it does open up typing issues² that need to be worked out in details before proceeding with a full blown proposal.

References

- [1] H. Abelson, J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, second edition, 1996.

²`lazy:call` would need to know the actual resulting type of the argument expressions to meaningfully set up a discrimination for the underlying method.

- [2] A. Baine. Funds: Functional Data Structures in Common Lisp. Project page at <http://common-lisp.net/project/funds>, 2007.
- [3] S Burson. FSet: a functional set-theoretic collections library. Project page at <http://common-lisp.net/project/fset>, 2007.
- [4] M. Lamari. Heresy: Haskellesque lazy-list and functional tools with a Common Lisp slant. Project page at <http://cl-heresy.sourceforge.net/Heresy.htm>, 2007.
- [5] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
- [6] K. M. Pitman. The Common Lisp Hyperspec. published online at <http://www.lisp.org/HyperSpec/FrontMatter/index.html>, 1994.

(WORK-IN-PROGRESS...)



JERRY BOETJE
DAVID WILLIAMS
ROBERT SHIELDS
HECTOR RAPHAEL MOJICA
SETH RYLAN GAINNEY
College of Charleston

Background	3
Introduction	3
What is CLforJava?	3
About The Author and Lisp	3
History	4
Becoming an Educational Exercise	4
Phases by Semester	4
Basic Architecture	6
The Intertwining Imperative	6
Type System	6
Symbols and Packages	7
Cons and List	7
NIL	7
Functions	8
Closures	8
Interesting Components	9
The Compiler	9
Transfer of Control	9
Pathnames and Abstract Streams	11
Defstruct	14
Documentation	15
Handling Load-Time-Value	15
Hashtables	15
Support for CDR-5	18
Near-term Futures	19
Unified Printing Architecture	19
Integrating Existing Components	19
Unicode 5	19
Sequence Functions	19
Non-Simple Type Specifications	19
Going Open	20
It's Time	20
Building the Plan	20
Executing the Plan	20
Call for Support	20
The Remaining Big Ones	20
Ones we know we can do	20
New Compiler	20
CLOS	20
Accessing Java	20
Ones we're not so sure about	20
Continuations	20
Debugger	20
Summary, Acknowledgments and References	22
Summary	22
Acknowledgments	22
References	22



Background

1. Introduction

The CLforJava project started in 2002 as a vehicle for advanced students to extend their computer science education by tackling difficult architectural and programming problems in the context of a large, complex product. Before it started, the faculty asked me to use this project to revamp the capstone software engineering course. The aim was to give the students the experience of working on a very large, multi-semester project akin to the environment they would encounter in their professional lives.

2. What is CLforJava?

Common Lisp for Java is a project used to simulate, in the classroom, the “real-world” environment of modern software development. The class is the capstone Software Engineering Practicum, where the students work on a large project using the tools and processes they are likely to encounter in their professional career. The long-term goal is to create a new, ground-up implementation of the Common Lisp language¹ running on the Java Virtual Machine. Its primary architectural and implementation goals are:

- Full compliance to the ANSI specification
- Execution on the Java Virtual Machine without translation to the Java language
- Transparent interaction (intertwining) between the Common Lisp and Java languages without use of a Foreign Function Interface (FFI) or syntactic sugaring.

The CLforJava project is also a research vehicle for talented undergraduate and Master’s-level graduate students to architect and implement a complex subsystem such as a compiler, defstruct, the core of CLOS, and Java-based documentation system.

3. About The Author and Lisp

In the period 1984/5, I was a developer in the VAX Lisp project at Digital Equipment Corp. (DEC). My first assignment was to design and implement the VMS FFI. While it was a success, the inherent anti-elegance of the solution tainted my view of all such interfaces. It was my next assignment - build a programmable editor for VAX Lisp - that (accompanied with some level of pain) taught me about Lisp - or so I thought. After the product delivery, I entered a Master’s AI program at Brown University under the tutelage of Eugene Charniak who nudged me into a fuller understanding of the magic of Lisp. Lisp thinking navigated me through challenging projects in Lotus, Sony, and several small start-up companies. In 2002, I stepped back from the front lines of coding and took a position in the Computer Science department at the College of Charleston. It was there that my interest in (and passion for) Lisp took an unexpended turn.

¹ As defined by the ANSI Common Lisp specification 1989
College of Charleston



History

1. Becoming an Educational Exercise

In my second teaching semester, I thought of reviving my idea of building a version of Common Lisp running on the JVM and transparently accessible to/from Java without an FFI. The plan was to start a research project using a few of the best CS students for one or more semesters to build and integrate components of the system. In some discussions with a colleague, he suggested that I take over the capstone Software Engineering course, using the Lisp project as the anchor, and run it as an industrial project using industrial tools and processes. The faculty concurred, and we started in Fall of 2003.

2. Phases by Semester

The project is broken into semester-size chunks of work that students, as a team, build, test, and integrate. They must first learn to effectively use the industrial-level tools such as Perforce, netbeans, TWiki, and Bugzilla. For most of them, this is their first encounter with a very large and complex system with tens of thousands of lines of code in multiple languages. They learn to follow the procedures and to use the tool suite to manage the complexity.

In addition to this course, a number of talented students continue to work on some of the complex components as single-semester independent studies or Bachelor's and Master's theses.

Here are the components by semester. Some carry-over to additional semesters in the case of thesis work. The choice of the semester tasks is determined not only by the "logical" order of components but also by the size and capabilities of the particular class.

SEMESTER	COMPONENTS	THESIS / INDEP STUDY
F2003	<ul style="list-style-type: none">• Basic Type System• Symbols• Basic arithmetic• Basic REP loop• Simple Reader / Printer	<ul style="list-style-type: none">• Bootstrap compiler
S2004	<ul style="list-style-type: none">• Lambda forms (required args only)• Package system• Stream system• Macros	<ul style="list-style-type: none">• Bootstrap compiler• Reader
F2004	<ul style="list-style-type: none">• Characters• Unicode Integration• Constants, Variables, and Keywords	<ul style="list-style-type: none">• File compilation• Loader
S2005	<ul style="list-style-type: none">• Environment• Basic Printer	<ul style="list-style-type: none">• Update compiler with environment
F2005	<ul style="list-style-type: none">• Simple Strings• Simple Arrays• Comparisons	<ul style="list-style-type: none">• Update compiler with environment
S2006	<ul style="list-style-type: none">• Arrays• Strings	<ul style="list-style-type: none">• Non-positional numbers
F2006	-- Hiatus --	<ul style="list-style-type: none">• CLOS MOP core• Lambda-list parsing in Lisp



SEMESTER	COMPONENTS	THESIS / INDEP STUDY
S2007	<ul style="list-style-type: none"> • Transfer of control • Bits and Bytes • Complex Numbers 	<ul style="list-style-type: none"> • CLOS MOP core
F2007	<ul style="list-style-type: none"> • Pathnames • Abstracting stream • HTTP stream 	<ul style="list-style-type: none"> • Lambda List Parsing
S2008	<ul style="list-style-type: none"> • List functions in Lisp • Hashtables - has support for CDR-5 	<ul style="list-style-type: none"> • New compiler in Lisp

Due to the downturn in CS enrollment (experienced by most all colleges and universities in the US), the practicum course will run only once per year for the next few years.² With this change comes opportunities. I now teach the pre-requisite software engineering theory class in the Fall 2008 semester. Instead of doing the usual dry lecture, the course will devise a plan for moving the project into the Open Source community. The plan will be executed by the same students in the following semester as part of the practicum course. From then on, the project will be driven not just by the needs of the course but also by the needs and desires of the community.

² The enrollments are now increasing, but it will take a few years to get to the senior level.



Basic Architecture

1. The Intertwining Imperative

One of the basic tenets of the project is to create a system that is easily accessible from Java and vice versa. This rule has a pervasive effect on the architecture and the implementation. For example, a routine passes a lambda expression to a Java method. How does the method determine that this is a Lisp function? And having done so, how does the method apply the function to arguments? On the other hand, how would Lisp catch a Java exception? For that matter, how does Lisp know that some object is not a Lisp type? How is it possible for Lisp to deal with Java streams that are specialized when Lisp does not specialize streams? Lisp streams may have differing behavior, but the behavior is apparent only when queried or tried. (perhaps generating a runtime error). When should some Lisp types should be genericied? When should they implement interfaces appropriate to the use in Java, for example the Lisp List type implementing the `Collection` framework?

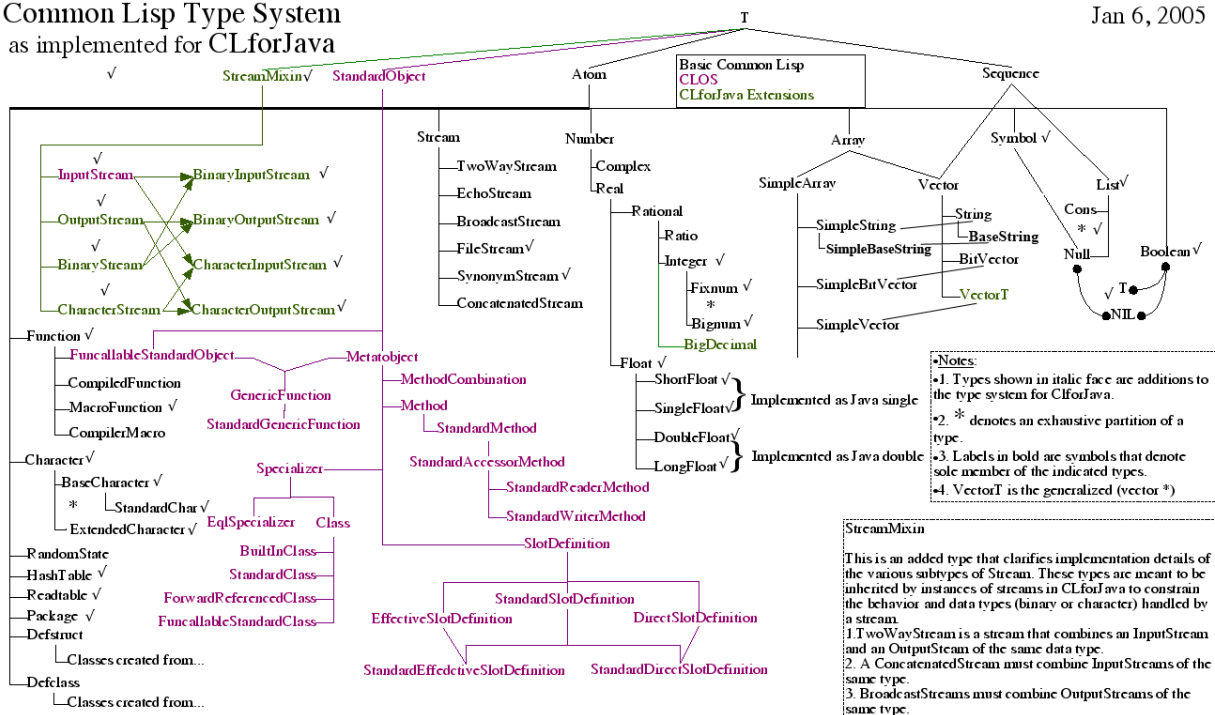
We determined that there were 3 components that would set the basis for the rest of the architecture:

- Type system
- Functions
- Symbols

2. Type System

Common Lisp Type System as implemented for CLforJava

Jan 6, 2005



The mapping of the Lisp type system onto Java is the core component that influences every other component. The project dubbed it the Rosetta Pattern, the key to melding the two languages. For all of its importance, it is a deceptively simple design.

The Common Lisp type system is a “tangled” web. Therefore it is not possible to mimic using Java classes. However, the Java interface component can be marshaled to mirror the non-tree structure of Lisp. But to making the intertwining of the types, a type interface must also carry a Lisp type name and the Lisp name must refer to the interface. This is handled by creating a static field in the interface that holds the symbol naming the Lisp type. That symbol in turn carries a reference to the actual Java interface object.



```

public interface Integer {
    public static Symbol typeName =
        Package.CommonLisp.intern("INTEGER");
    static { .to store interface in symbol. }
}

public interface Fixnum extends Integer{
    public static Symbol typeName =
        Package.CommonLisp.intern("FIXNUM");
    static { .to store interface in symbol. }
}

```

From this code, a Java programmer can determine if an object is a **Fixnum** by using the Java `instanceof` instruction. Likewise, when using a **TYPEP** function, for example `(typep 12 `fixnum)`, the function extracts the **FIXNUM**'s interface and the class of **12**. Then it can use the Java Class `isAssignableFrom` method to determine the type relation. A Java programmer can also determine if an object is a Lisp object by using `object instanceof T`.

By implementing Lisp types in Java interfaces, we can provide a system for creating instances of any instantiable type.³ Java interfaces may nest other interfaces and classes. Here we insert a nested factory class in any instantiable type. The Java factory has one or more `newInstance` methods that will return a new instance of the type.

For example, to make a **Bignum** a programmer calls `Integer.Factory.newInstance("123456789012324")`;

By using interfaces as type designators, we also take advantage of an interface's primary function - to specify the signatures of any methods required in implementing classes. For example, the **Number** interface that defines the corresponding Lisp type specifies the arithmetic operations that can be applied to all numbers. For example, to make a subclass of **Bignum** called **Infinity**, the Infinity interface defines the Lisp type (**INFINITY**). But the **Number** interface defines what methods the implementing class must code.

3.Symbols and Packages

Symbols are a simple structure in Lisp although they do have many attributes. They are easily implemented as a Java class. However, the various uses of symbols in Common Lisp lead to some interesting wrinkles. Some examples:

- T is a symbol whose value is constant and is itself. And it must be loaded before other types can be loaded.
- The binding stack for a symbol is implemented in the symbol itself. The obviates the need to keep a separate binding stack that must be searched.
- There are a number of symbols defined in Common Lisp (aside from the function and type symbols). These may be variables or constants. Since they are used so often, these symbols are accessible to Java as fields in well-known interfaces by their usage: **Variable**, **Constant**, and **SpecialOperator**.
- Those symbols that name Common Lisp special operators are also of type **SpecialOperator** allowing for use of the `instanceof` instruction in the compiler.
- Some symbols are defined to hold values of only one type, for example `*package*`. They are special subclasses of **Symbol** that will reject any attempt to set it to an illegal type.
- NIL is so weird, it merits its own section...

Packages are implemented by subclassing a Java **HashMap** object. They also carry fields for **USES**, **USED_BY**, **SHADOW**, and **EXPORT** operations.

4.Cons and List

Cons is the simplest of the data structures. In our implementation, a **Cons** has generic parameters for the `car` and `cdr`. It is of course a subclass of **List** in both Lisp and Java. **List** is a superclass of **Cons**, and it's factory methods have a generic `car`. Also, the **List** type implements the Java **Collection** interface, supporting the usual Java access to lists.

5.NIL

NIL is the oddest of the objects in the system. It is both a **List** and a **Symbol**. It is a singleton. And must be loaded very early in the startup phase. If we were to leave loading up to the Java on-demand loader, it's guaranteed to get 2 instances of NIL. That would not be the best of all worlds.

Our initial designs called for either a **Symbol** class that implemented the **List** interface or a **List** class that implemented the **Symbol** interface. Neither of these solutions worked. What we needed was a class that sub-classed 2

³ Many types are abstractions of collections of types. For example, **Atom** is not instantiable but **Symbol** is.



classes - forbidden by Java. Our solution was to create 2 classes, each of which is both a Symbol (**NilSymbolImpl**) and a List (**NullImpl**). In each class initialization, there is a reference to the other. The order of the code is such that the initialization order of the 2 is fixed (not at the whim of the Java class loader). The last one initialized (**NullImpl**) uses an instance of the other as a delegate for the Symbol component. It also wins the race to be in the Common Lisp package. It is rather reassuring that the Java class loader knows how to load **T** (the topmost type) and create a plist in **T** with value **NIL** before the end of the initialization of **T** - a superclass of **NIL**.

6.Functions

For each unique function (each **lambda** definition), there is at one instance of a unique Java class implementing the function's code. These classes all implement the **Function** interface which defines an **apply** method that takes a **List** of arguments and returns an **Object**. If the number of parameters are known at compile time, the class may implement one or more **FunctionN** interfaces each of which have a **funcall** method with that number (**N**) parameters. When the compiler encounters a lambda form, it compiles the form and arranges that an instance of that class is deposited in the code stream. It is possible to define the Java name of the class with the **system::%java-class-name** declaration.

For example,

```
(lambda (list)
  (declare
    (system::%java-class-name "First"))
  (car list))

class First implements Function, Function1 {
  Object apply (List args) { .... }
  Object funcall (Object arg) { ... }
}
```

7.Closures

The current method of closures is correct but not very efficient. To each lambda, the compiler allocates an array of objects that represent the visibly closed variables in the lambda. This array is embedded in an object that becomes part of the runtime tree structure of the lambdas in the program. Every lambda, whether it closes a variable or not, has an instance of the closure class. These are created when an instance of a lambda is created and mimics the tree structure of the runtime. A function has direct access to its closure set, and the closure set records its parent. When the function requires access to a closure set, the compiler has calculated the number of hops (0 being the local closure set) up the parent links and the array index holding the current value of the closure.

The project plans to re-write the compiler, bringing it up to current techniques and coding in Lisp. The issue of closure implementation will be part of the research goals.



Interesting Components

1. The Compiler

The compiler has undergone 3 distinct phases: bootstrap, environment-based, and modern in Lisp. The first two are coded in Java - even though they use Lisp constructs. The third will be the basis of a modern compiler, using interval analysis to perform control and data flow analyses and will be written entirely in Lisp.

Having already determined not to create a separate interpreter, we required a compiler that could perform at least the basic transformations required to compile simple function application: **lambda**, global symbols, global binding, **if**, **progn**, **quote**, lists, numbers, **let**, and **setq**. The bootstrap compiler was a 2-pass: semantic analysis and code generation - both very simplistic. The code generator used the Oolong JVM assembler to create an array of bytecodes suitable for a Java class loader. The only upgrade made with this compiler was to support file compilation and file loading.

Near the end of the second year, we upgraded the compiler by adding an environment and upgrading the compiler to handle local and closed variables. The other major alteration is to switch from Oolong (text-based assembler) to the ASM facility (API directly to bytecode) from Objectweb. We achieved a 60-times speed up in the compilation process and the ability to add new Java 5 facilities such as annotations.

This summer we expect to have the first milestone in the new Lisp-built compiler: the application of interval analysis to the fully expanded code. This will also entail rebuilding the environment system for speed and additional functionality. The new environment will also carry binding information for locally-named functions (**labels** and **flet**). Near the end of the summer, we will assess which control and data flow transformations to implement. At a minimum they must include closure detection, register allocation, and local **tagbody/go** optimization.

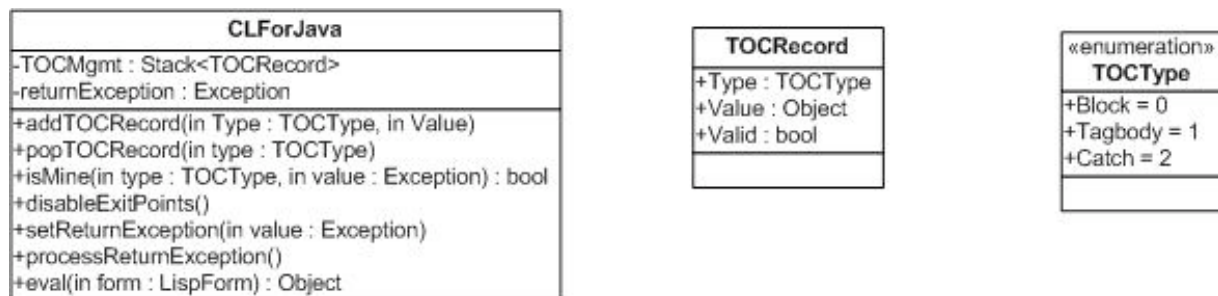
The next phase in compiler development is cognizance of data types. It should be capable of generating code for testing forms for adherence to the data declarations. It should also be able to generate optimized code based on data type, e.g. using Java **int** arithmetic with **fixnums**.

2. Transfer of Control

For the three types of TOC: **TAGBODY/GO**, **CATCH/THROW**, **BLOCK/RETURN-FROM**, and **UNWIND-PROTECT**, we implemented a unified mechanism using the Java **try/catch/finally** mechanism. This involving creation of specialized Java Exception classes that hold return values (catch and return-from) and a stack of markers used to implement the exit points.

Class Diagrams

Static diagram and descriptions



Attribute Additions to CLForJava

TOCMgmt - This is a runtime stack that manages all of the runtime transfer of control points. During compilation, all TOC types establish the code to manage the pushing and popping of TOCRecords to/from the stack. The push will always occur immediately in the code and the pop will always occur in the finally block of the corresponding TOC.



returnException - If the unwind-protect block is entered because an exception was thrown (ie...Go, Throw, or Return-from occurred) and not from regular control flow returns, it must store and rethrow the corresponding exception after it has executed the cleanup form. This is the exception that will be rethrown at the end of the cleanup form (if one is going to be rethrown at all).

Method Additions to CLForJava

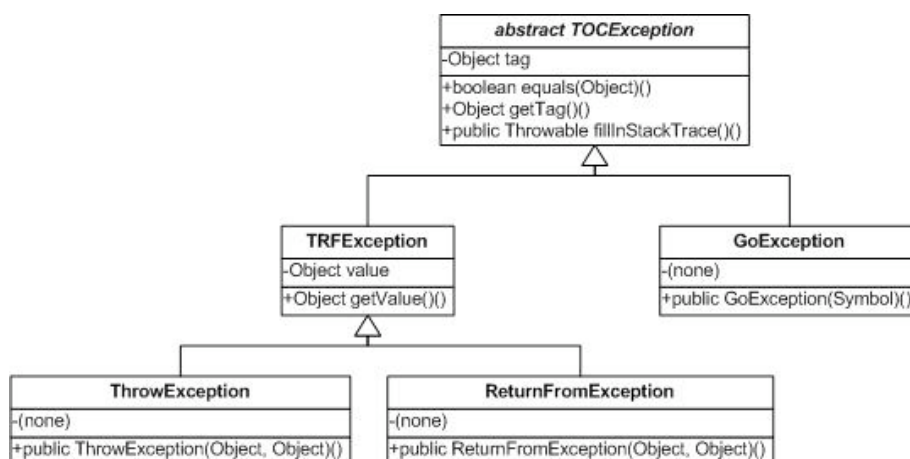
addTOCRecord - This method is a run-time registration mechanism for all of the TOC special operators. All TOC bodies (block, tagbody, and catch) immediately call this method at runtime when their scope is entered. They store their type of TOC and what tag values or symbols they are authorized to process in the event a corresponding event (go, throw, return-from) occurs.

popTOCRecord - All of the TOC special operators will generate code in their finally block to call this method. This is necessary cleanup in order to ensure that the runtime TOC stack is correctly maintained. Since the TOCMgmt stack can have heterogeneous TOC types on the stack at any given time, the callers will pass the type and this method will find and remove the first instance of that corresponding type on the stack.

isMine - This method is only called if one of the TOC types actually caught an exception; otherwise, the finally block of the corresponding code will be executed and the TOCRecord will simply be removed from the TOCMgmt stack. If a handler does catch the exception, it will call this method with the corresponding TOC type and exception. This method will find the first instance of the corresponding type on the stack, verify that the Exception is of the correct type and if so compare the value with the value that this already on the stack (ie...what tags/symbols this exception is allowed to process) and return the result of the comparison. If the compare fails, the caller will remove himself from the stack and rethrow the exception; otherwise, they will remove themselves from the stack, handle the exception, and continue as normal.

disableExitPoints - Immediately after the finally block from the unwind-protect form begins execution, this method is called. This method will traverse up the TOCMgmt stack and disable every single TOCRecord until it finds the first instance of a TOCRecord that matches the current returnException value. All other methods (such as isMine) will always check the valid bit before validating an entry in the TOCMgmt stack. This will ensure that the cleanup form from unwind-protect is unable to transfer back into anything that was within scope during the protected form execution. If this does occur, when the exception handler from the protected form tries to handle the exception, it will get a false from isMine and therefore, throw the exception right back up the runtime stack.

setReturnException - This simply sets the returnException value in case the TOC occurred because of an explicit control transfer. This value will be utilized later to throw back up after the TOC has executed finally code. If another control transfer occurs in the finally code, the value will be ignored and never used. It will get set back to null every time a processReturnException call is made.



processReturnException - If an exception was caught (and was not handled by the current TOC), it would have been stored as the returnException. If this was the case, once the finally code of the TOC is completed, this will be the



last call made before going on to the continuation block. If there is a valid exception, it will simply be rethrown; otherwise, control will return to the current TOC, and they will proceed to their respective continuation block.

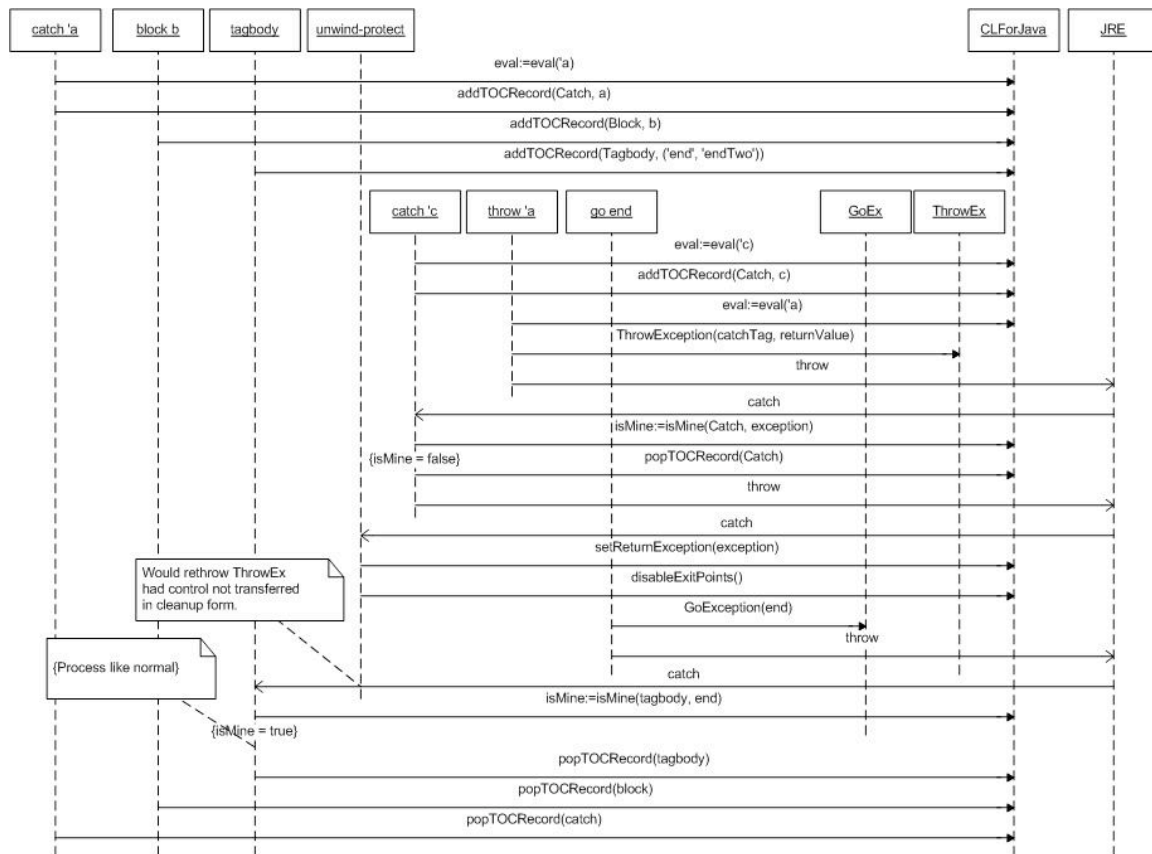
TOCRecord - This is simply a struct that holds the information about an instance of a control transfer that will be held on the TOCMgmt stack.

Type - This is an instance of an enumeration TOCType {Block, TagBody?, Catch, UnwindProtect? } and simply store the type of the TOC record on the stack.

Value - This is the tag or symbol that the corresponding type is allowed to process (for catch 'a, it would be 'a'; for block here, it would be 'here', and so forth). For tagbody, since it can have multiple labels, it will be a list of objects holding all of the corresponding label values. That is required so that CLForJava? can effectively answer the isMine question.

Valid - This value is defaulted to true and is only set to false when an unwind-protect protected form is disabled after the cleanup form is entered.

Runtime TOC Sequence Example



3.Pathnames and Abstract Streams

Pathnames in CLforJava have been designed to handle not only traditional local filenames but also any resource described by a URI. Core support for URIs was added without extensions to the language or exceeding the bounds of the CL specification. During design, the key insight lay in recognizing the concept of a “meta-device” as defined by URI schemes. URI schemes such as “file”, “http” or “mailto” specify how a given resource is structured and accessed. More importantly, they provide a namespace, which enforces the uniqueness of a given identifier. Similarly, the device component of pathnames specifies a physical or logical storage area in which each resource has a unique



path. Therefore, our pathname implementation abstracts the device concept to include all possible URI schemes, represented as a keyword in the device component.

For the base case of local filenames, there is the URI scheme “**file**.” As shown in Table 1, any pathname representing a local file will have the keyword `:file` as the value of its device component. The name and type is parsed in the obvious manner and the version component is always set to `:UNSPECIFIC`, following the example of Allegro CL. The directory component is quite similar to other implementations. However, some file systems have chosen to separately name a physical device or volume (e.g. drive letters in Microsoft file systems). It was decided that such information, if present, would be included as part of the directory list. This seems reasonable since such a device is certainly part of the hierarchical path uniquely identifying a file. This strategy also offers more consistency between filenames on Windows systems and those on UNIX-based systems. This is convenient for a multi-platform implementation of Common Lisp such as CLforJava.

Namestring	"C:\\foo\\file.txt"	"foo/bar/"	"/foo/.hidden"
Components			
Device	:file	:file	:file
Host	NIL	NIL	NIL
Directory	(:ABSOLUTE "C" "foo")	(:RELATIVE "foo" "bar")	(:ABSOLUTE "foo")
Name	"file"	NIL	".hidden"
Type	"txt"	NIL	NIL
Version	:UNSPECIFIC	:UNSPECIFIC	:UNSPECIFIC

Table 1: File based pathnames in CLforJava

URIs with other schemes are also easily handled. Example 1 shows the mapping of an http-based URI to pathname components. The query part was stored in the directory list.

Example 1 – (pathname “<http://www.cofc.edu:8080/foo/index.htm?query=x>”)

```

Device      :http
Host        "www.cofc.edu:8080"
Directory   (:ABSOLUTE "foo" "?query=x")
Name        "index"
Type        "htm"
Version     :UNSPECIFIC

```

Syntactically a URI is identified as opaque if it is absolute (specifies a scheme) and its scheme-specific-part does not begin with a forward slash ('/'). Opaque URIs are not subject to parsing beyond what is stated above. Non-opaque URIs are termed 'hierarchical'. With any hierarchical URI, the scheme-specific-part may be further parsed according to the syntax

[scheme:] [//authority] [path] [?query] [#fragment]

and the *authority* component may be further parsed as

[user-info@]host[:port].

CLforJava takes advantage of this hierarchy by collapsing a URI's nine possible components into only five: scheme, scheme-specific-part or authority, path, name, and type. The resulting parsing strategy implemented by CLforJava is summarized in Figure 1 and discussed in detail below.



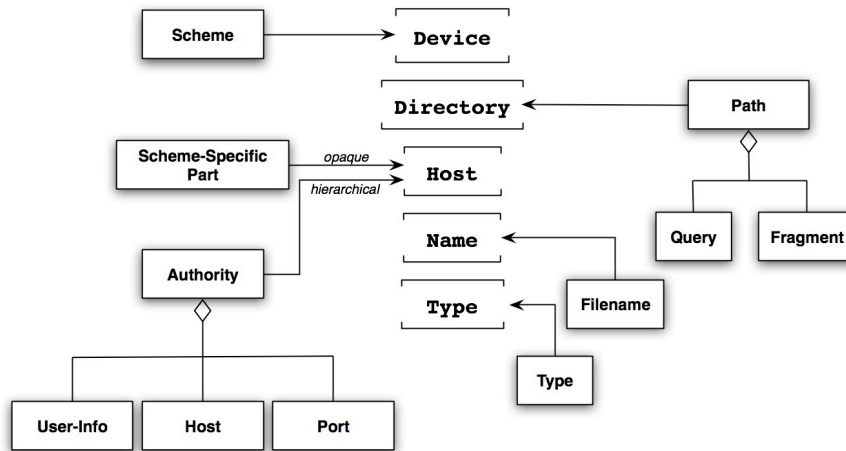


Figure 1 – URI component mapping

Either the scheme-specific part of an opaque URI or the authority part of a hierarchical URI is stored in the pathname's host component. Doing so preserves and encapsulates all the access information in an intuitive location. It also seemed logical to store the path information in the directory component of the pathname. Each element of the path therefore exists as an element of the directory list. Strictly speaking, the fragment and query parts of a URI are not subparts of the path. Since one of our derived requirements for pathnames was that we preserve syntax, it was unacceptable to add any new components to the pathname type. Therefore, we chose to treat the fragment and query as part of the path so that they can be easily stored in the directory list. Due to the syntax of URIs, these items are easily identified within the list (queries begin with '?' while fragments begin with '#'). Some URIs may specify a specific file such as the common "index.htm" in an HTTP-based URI. URI syntax does not explicitly provide separate slots for the name and type of such a file; it is simply included as part of the path. However, since pathnames have components for identifying the name and type of a file, it seemed reasonable to utilize them. CLforJava will parse the path of a hierarchical URI, retrieve the name and type information, if any, and store it appropriately.

While a pathname carries the specification of a connection mechanism, the constructed conduit must implement the connection in the form of a stream. Common Lisp defines character or binary transfer with the stream being limited to simple, unstructured sources and sinks of data. With the advent of the Internet and URIs, the stream must become a more adaptable agent in the transaction. As with the pathname, the stream must transform into an abstract entity that can morph into a scheme-specific stream as required by the pathname.

When the basic pathname is abstracted, the basic stream must provide a matching abstraction layer. The **OPEN** function now delegates common abstractions between the pathname/stream pair. It is the responsibility of **OPEN** to locate the concrete stream implementation based on the type of pathname and return an appropriate active stream, or signal an error. As with the file streams, certain options or combinations of options are invalid. For example, an **HTTP** stream supports directions of **:INPUT** or **:IO** but not **:OUTPUT**.

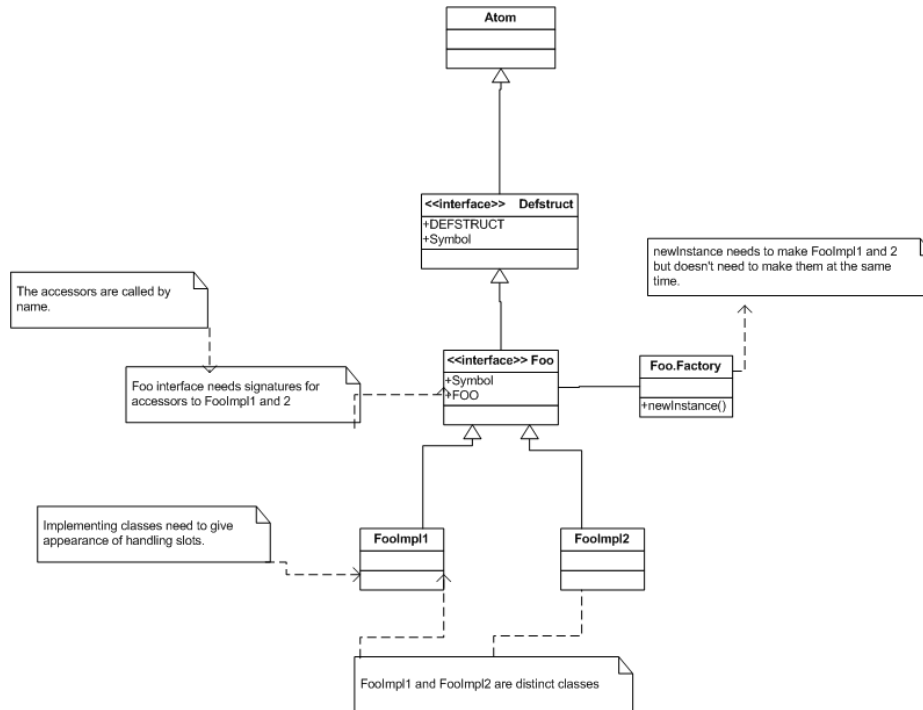
These added stream types should preserve the semantics common to file-stream types. To use character-based http-stream types as an example, the **READ** functions return characters or strings, and **FILE-WRITE-DATE** and **FILE-LENGTH** return the same metrics as they would for a file-stream type by gathering this information from **HTTP** headers. Write functions act as a request sender moving **POST** data.

Currently, CLforJava supports only **HTTP**-stream types. Future additions, just like the addition of **HTTP**-stream types, must include the proposed stream, the meta-device it is related to, and the basic functionality of the stream (such as reading and writing) within the added code. Like any responsible architecture, CLforJava requires no retroactive changes to other stream functions such as **READ-LINE**.



4.Defstruct

DEFSTRUCT is implemented in CLforJava using the following UML diagram. Descriptions follow:



- **defstruct.lisp** -- This is the macro definition and a lot of helper functions that process all the arguments to DEFSTRUCT and properly expand the various functions
- **lisp.system.compiler.IntermediateCodeGenerator.java** -- the ICG defines a special operator, %DEFSTRUCT, which handles setting up new struct definitions by generating code directly for the JVM to create new classes for each struct definition. During this class creation the ICG ensures that certain fields, such as typeName, are properly set for each new struct definition.
- **lisp.extensions.type.Defstruct.java** -- the top level interface that all structs implement. Each new struct definition causes the generation of a new interface inherited from this one (done in the ICG).
- **lisp.system.DefstructImpl.java** -- the superclass for all struct instances. Each new struct definition causes the generation of a new implementation class which extends this (done in the ICG).
- **lisp.system.function.MakeDefstructInstance.java** -- the function which creates new instances of structs
- **lisp.system.function.GetDefstructSlot.java** -- the function that gets the value of a struct's slot
- **lisp.system.function.SetDefstructSlot.java** -- the function that sets the value of a struct's slot
- **lisp.system.function.CopyStruct.java** -- the function which is called by the copier function expanded by the macro. DefstructImpl implements the Cloneable interface and overrides Object.clone() allowing our copier function to work.



5.Documentation

While we are aware of several new Lisp documentation systems, none of them provide the kind of flexibility afforded by the Java resource system in terms of translation, localization, and formatting. Since we are also not fixed on Lisp solutions to all problems, a student devised a documentation system that fits within the existing Lisp **DOCUMENTATION** function and the common translation and formatting processes.

The student added code in the compiler to gather doc strings and other information such as argument types. He also added a **SETF** function for **DOCUMENTATION** that lets the compiler harvest that information as well. All of this data is turned into (heavens!!) **XML** format. For **COMPILE** compilation, the gathered information is preserved in class annotations. Information gathered by the file compiler is also stored as annotations. Then, as a last pass over the compiled code, the compiler gathers up the **XML** and adds it to the jar file. When that file is loaded, the doc is also loaded and made accessible through **DOCUMENTATION**. The documentation is processed via an **XSL** transformation depending on the type of display: simple text, **HTML**, **PDF**, etc. Furthermore, since we use Java resources, we can rely on the Java resource system to locate a localized version of that text.

6.Handling Load-Time-Value

One of the advantages of creating classes for functions is seen in our implementation of the **Load-Time-Value** special operator. Since we control the structure of the lambda class, we can control the time at which functions are evaluated. In this case, if we use the CLtL2 example, **(load-time-value (first *my-array*))**, we would first add a **static final** field to the current lambda class being built. Then the compiler wraps the load time form in a no-argument lambda, ex **(lambda () (first *my-array*))** and compiles that lambda. As with any lambda, the compiler creates the implementing class, but the instance of the class is arranged to be placed into the created static field. One of the tasks of the code generator when it encounters the original class (the one containing the static field), it creates the code for the Java class initialization. It also adds code to the class initialization to evaluate the function instance in the static field. It then places the result value back into the static field. Since it is a **static final**, it cannot be changed later. Effectively, the load time value is evaluated as a side-effect of the function's class loading. Access to the value is a static field access - a very fast operation in the JVM.

While the use of a static final field prevents alteration of the field, there is currently no protection for altering the contents of the object in the field. This will be dealt with in a new compiler.

7.Hashtables

Common Lisp hashtables are more interesting and sophisticated than those built-into Java. Java provides the equivalent of **EQ** and everything else is their Java **equals** and **hashCode** methods. The **EQL**, **EQUAL**, and **EQUALP** functions require different algorithms in their comparisons. Effectively, the Java **hashCode** and **equals** methods must change depending on the type of hashtable. But a goal of the project is that Java programmers have access to Lisp features transparently - including interesting hash tables.

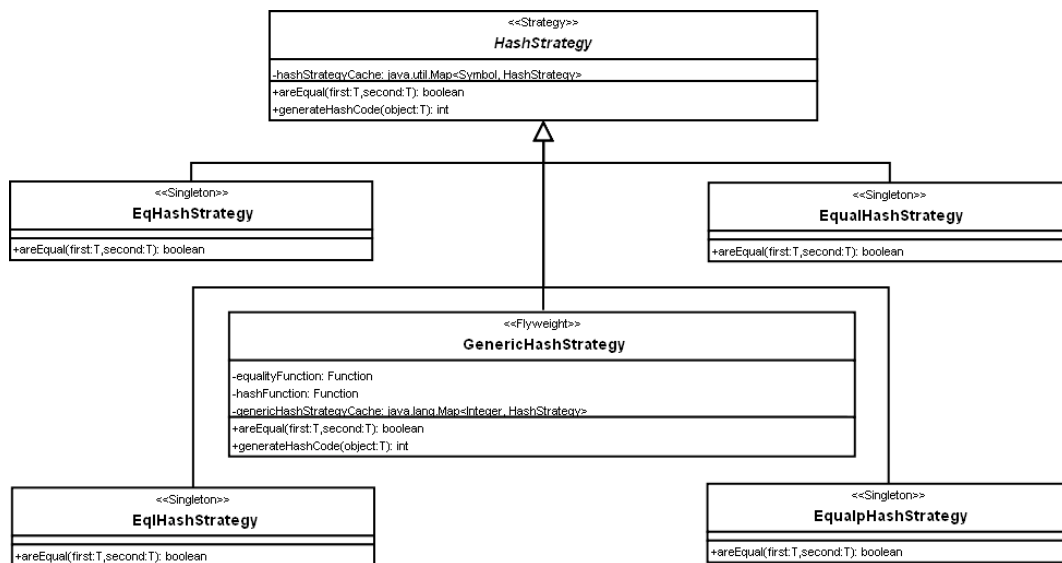
Our solution is rather elegant, involving attaching information to the classes of these data types in the form of Java annotations.



HashTable: Follows the standard convention in CLforJava of having Common Lisp types represented by Java interfaces. Contains a factory class used to instantiate new concrete instances of a hash table implementation. Declares commonly used hash methods, such as getters and setters, and operations to determine the size of the hash table, remove a key/ value pair from the hash table, and clear all elements from the hash table. Further, it defines the public constants representing default values for hash table implementations, such as default equality test, size, rehash threshold, and rehash size. Extends `java.util.Map<K, V>` in order to function in the same role as any other hash table implementation would in Java code. In effect, this is the Decorator pattern as laid out by the [Gang of Four](#).

IncludeInHashCode: Meta data, implemented as a Java annotation, that marks a class' fields as significant in calculating a hash code for an instance of that class. Optionally, an "order" can be specified, marking which field should be process first, second, etc., when calculating the hash code for an object of that class.

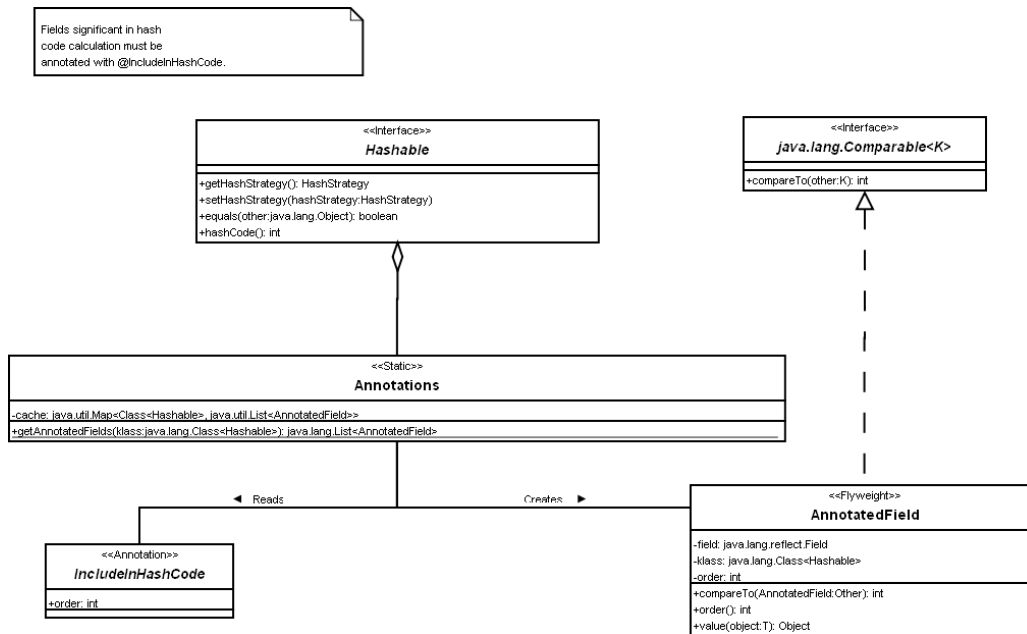
HashStrategy: Encapsulates the behavior required in order to properly store an object in a hash table based off of the four Common Lisp equality tests of `EQ`, `EQL`, `EQUAL`, and `EQUALP`. Through the use of metadata, an instance of this class is able to calculate a hash code for any object that implements `Hashable`, without any prior type information. This allows `HashTableImpl` to use any of Java's standard hash tables for its backing implementation and eliminates the need to implement a redundant `equals()` method in every CLforJava type. `HashStrategy` is an abstract class, which through its factory method returns a concrete Singleton implementation based off of the requested test type. The default hashing algorithm is based off of the material presented in [Effective Java](#). Because all subclasses of `HashStrategy` are private, only its interface is known to client code. This allows for easy extension by simply creating new subclasses of `HashStrategy`; unlimited definitions of equality or hash code algorithms can immediately be used in any class that implements `Hashable`. An initial implementation of `GENHASH` has been started by creating a `GenericHashStrategy` class that delegates both its equality test and hash code generation to supplied Function objects, which in effect creates another Strategy pattern layer.



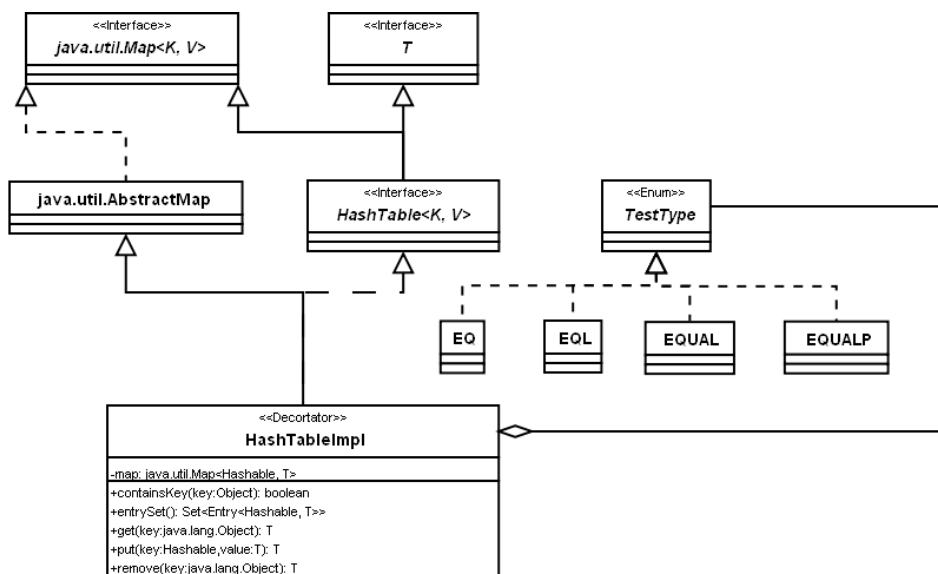
Hashable: Defines the contract between implementing objects and concrete implementations of `HashTable` necessary to ensure that each object used as a key in a hash table has the ability to generate a hash code and is testable for equality under Common Lisp's four different equality functions. Each implementing instance of `Hashable` must contain a `HashStrategy` object. This interface declares two primary methods to generate a hash code for the given object and to test its equality against other objects of the same type. Both of these methods are intended to delegate responsibility to a `HashStrategy` instance. Further, classes implementing this interface should include annotations through `IncludeInHashCode` to mark fields as significant in hash code generation. This interface defines a non-instantiable static class, `Annotations`, which gives access to fields annotated with `IncludeInHashCode` through a



list of immutable **AnnotatedField** objects. **Annotations** implements the Flyweight pattern, maintaining a cache of **AnnotatedFields** keyed by `java.lang.Class` objects, reducing the runtime cost of reflection.



HashTableImpl: Concrete implementation of HashTable defining the operations declared in that interface or inherited from `java.util.Map<K, V>`. It extends `java.util.AbstractMap` to reduce the amount of redundant code. Internally it uses a reference to a `java.util.Map<K, V>`, allowing the use of any of the standard hash table implementations, a third party implementation, or one written by CofC, by that hash table implementing, or being wrapped in a class that implements, `java.util.Map<K, V>`. Depending on the equality test, the actual backing `java.util.Map<K, V>` is either a `java.util.IdentityHashMap` (EQ) or a `java.util.HashMap` (everything else). Upon a **Hashable** object being placed within the hash table as a key, **HashTableImpl** sets that object's **Hash-Strategy** instance to the one appropriate for the requested equality test type.



8.Support for CDR-5

GENHASH is a possible extension to the Common Lisp specification (see CDR-5) supporting arbitrary equality tests and hashing functions. From the use of the Strategy pattern, this extension requires only additional, not altered, code. **GENHASH** requires that a pair of functions, consisting of an equality test and a hashing algorithm, be registered, and associated with a **Symbol**, to ensure that they produce valid results (which is left as an exercise for the reader). Once registered, hash tables can be constructed using this combination of functions.

Since **Hashable** objects don't work with concrete implementations, but rather the interface of the abstract **HashStrategy** class, this addition, from a Java standpoint, only requires making a new subclass of **HashStrategy** that behaves in the manner specified by the registered functions. The concrete child of **HashStrategy**, **GenericHashStrategy**, has been created to accommodate **GENHASH**'s requirements. This class includes a factory method that takes two **Function** arguments, indicating the desired test function and hashing algorithm. It overrides these behaviors in **HashStrategy**, delegating to these **Functions** through their **apply()** methods, and casts the results into Java primitive types. This allows for arbitrary additions to **HashStrategy**'s behavior dynamically at runtime.

GenericHashStrategy is currently implemented as a Flyweight object in order to avoid having the same equality and hash function pair be reproduced simply because they are associated with two different Symbols. Further, **HashStrategy** has been fitted with a **registerTestDesignator()** method that associates a **Symbol** with a **GenericHashStrategy** instance and a factory method that returns a **GenericHashStrategy** from an object pool keyed by a **Symbol**.

Future considerations include creating **EqualityFunction** and **HashFunction** interfaces to ensure proper argument and return types of the functions supplied to **registerTestDesignator()** and the creation of **GenericHashStrategy** objects, as well as eliminating the now legacy (after a less than a month!) **HashStrategy** factory method that takes an enumeration to determine test type.



Near-term Futures

1. Unified Printing Architecture

The current printer implementation is a patchwork of special code that evolved during the product development. Our intention is to meld a specific printer function to a type, much as a type carries a factory within the type interface. Faced with a Lisp object, the **WRITE** function would obtain the type's printer function and call it with the customary arguments. The type printer would use the various special variables to control the output. The printer for compound types (e.g. lists, structures) will recursively print the components as defined by their type printers.

There has been no significant architectural work done for this concept. Designing and implementing this concept would be a good candidate for an independent study or possibly a Bachelor's essay.

2. Integrating Existing Components

The CLforJava project does not intent to build all of a Lisp system from the ground up. Two examples are the **Pretty Printer** and the **Loop** macro. Code for both of these components exist in public domain, and we have no intention of re-writing these utilities. This is an appropriate semester's work for the software engineering course. The students would be presented with an existing code base and must determine the steps (including the implementation of needed components, e.g. the ***features*** subsystem) required to fully integrate and test these facilities.

3. Unicode 5

The current build of CLforJava implements the entire Unicode 3.1 character base. It also integrates the Common Lisp character system based on Unicode (ref 3). While the Unicode specification defines a number of algorithms for manipulating characters, Java 5 has implemented only the comparison algorithms. Java 6 provides more support for character manipulation and can be the basis of some additional Common Lisp functions to deal with the complexities of Unicode. Integrating CLforJava into Java 6 and implementing the Unicode support is the proper level of difficulty for a one-semester, independent study.

4. Sequence Functions

Implementation of the sequence functions is slated for the Spring 2009 semester of the software engineering course. As with the list functions implemented in Spring 2008, all of the coding work will be done in Lisp.

5. Non-Simple Type Specifications

The current type system is based on lisp atomic types such as **Fixnum** or **Vector**, mimicked very well with Java interfaces as described before. However, Common Lisp defines more complex types that may be created at runtime. These are the compound and compound-only types that define much more complex types. Some are straightforward:

- **AND** - a sub-interface of the set of types.
- **OR** - type relations create a "hidden" superclass of the set of types.

Simple constraints such as **(integer 0 10)** may be recorded in the **fixnum** type and can be referenced as type information for a variable or function. Implementation of the **satisfies** type constraint using the type interface pattern will undoubtedly entail some amount of cleverness. This is an excellent Bachelor's Essay project.



The Remaining Big Ones

This section discusses the remaining “big ones,” the set of facilities that are either required by the CL specification, desired by users, or specific to CLforJava and that set it apart from other JVM-based lisp systems. None of these are in the class “easy” or even the class “difficult”. They belong to the class “really hard.” Any of these would be appropriate for a Master’s thesis.

1. Ones we know we can do

1.1. New Compiler

The current compiler is Java-based, and has gone about as far as it can go. The basic design is a two-pass, non-optimizing compiler that does little control flow and almost no data flow analysis (just enough to handle closures). What is needed is a new compiler, written in Lisp, that do all the analysis and optimizations expected of a modern compiler. We have started the process by re-writing the code emitter in Lisp (also allowing to see the disassembly as a list of instructions). A student is currently working on a new compiler having a modern structure but not much in the way of optimization. Using interval analysis, we expect this version will compile code as well as the current compiler, but with better register allocation and **GO** handling. But it will be a platform for expansion of the capabilities of the compiler over time.

1.2. CLOS

This is the largest remaining component in CLforJava. While there is a Master’s Thesis defining the internal structure of the CLOS MOP (** Jay’s ***), there is an enormous amount of work remaining to build the full AMOP and from there the complete CLOS system. And once created, it must be integrated into the existing system. This is Master’s level work and will also require the work of other students to build and integrate components.

1.3. Accessing Java

The holy grail of the project is to build a system that can intertwine Lisp and Java seamlessly without “strange” constructs (aka FFI’s). While the Java->Lisp is well in place, the reverse requires more sophisticated components. The first of these, Common Lisp Java Packages, was described in (** Jerry **) and in section (** java packages**). Creating these components is not difficult, but integrating the Java object model into the CLOS structure and function is intricate, requiring, among of things, alterations in the compiler.

2. Ones we’re not so sure about

2.1. Continuations

If accessing Java is the holy grail, the implementation of continuations in CLforJava is the pinnacle of Lisp. Periodically, there are discussions in the Java community regarding continuations. There is currently a proposal to build closures into the language. From there, others may attempt continuations. Failing that, CLforJava may seek a solution using the Java exception facility. This is likely to be slower than just long-jumping, but will at least deal with the stack overflow problem.

2.2. Debugger

How can we work without one? Java provides a suite of facilities that may be stitched together to create a Lisp debugger for CLforJava. Some of these may prove to be simple adaptations of Java facilities. For example, proxies may support the **TRACE** facility. Building a true Lisp debugger is a more complex undertaking. The Java Platform Debugger Architecture (JPDA) provides a mechanism for monitoring and controlling a running JVM and would be the obvious platform for building a debugger. However, much of the information usually available from the JVM is heavily biased to the Java language. The Lisp debugger would require the compiler to liberally sprinkle annotations into the classes, methods, and fields that would provide sufficient information to create an effective Lisp debugger.



Going Open

1.It's Time

To date, CLforJava has been an open-source, closed-development project. It has a very good vehicle for training undergraduates in Software Engineering and, of course, building a cadre of Lisp-warped programmers. At this juncture, the project has met one of the critical milestones in the development of a programming language: it can begin to write it in itself. From here, the remaining "simple" features can be implemented relatively (relative to doing it in Java) quickly. To finish the job will take the work of seasoned Lispers (and Java programmers). These people are rare in a small CS department in a liberal arts college. So it's time to reach out to the Lisp community for help.

The focus here at CofC will change from being primarily a development organization to primarily a support and management role with some development. In many ways, this will serve the students' engineering education because they are entering a development environment that is very different from 2000. Accessing, assessing, contributing to, integrating, and managing Open Source projects are the direction of software in 2008 and beyond.

2.Building the Plan

So, having said we're going Open, how do we go about it? In this Fall, I'll also be teaching the Software Engineering theory course - where they get the book learning. While we will do some book learning, I will have them create a plan for opening the CLforJava project. My primary guide is Karl Fogel's [excellent book](#) on creating an Open Source project. One of their hurdles is to get reviews on their plan from experienced Open Source principals. The reviews will be a component of their grade, so they have some incentive to do a good job.

3.Executing the Plan

In the Spring 2009 semester, most of those students (and some others) will also be in the capstone course. They have 2 jobs in the Spring: implement the plan, and, time permitting, implement the Sequence functions. Here we would want external people to use our Open Source system and help find the bugs in the process (heaven knows there are quite enough bugs in the code!).

4.Call for Support

This is a plan for a plan and hopefully an execution on the plan, leading to CLforJava a known and active Open Source project. About 130 students have put a great deal of time and effort into something they didn't think they could build. To go beyond, we need help from the people reading this paper. If it intrigues you sufficiently, my e-mail is on the cover page.



Summary, Acknowledgments and References

1. Summary

The CLforJava project has proven its worth as an undergraduate research and teaching mechanism for Software Engineering. In the process, it has created an incomplete but working implementation of Common Lisp. The current product is a compiling Lisp system supporting most of the basic aspects of Common Lisp. Recent additions include DEFSTRUCT, Hashtables, all list functions, all forms of lambda list, and the beginnings of a modern compiler implemented in CLforJava. While the project will continue to add more required functions such as sequences, its major transformation is to a true Open Source project with the attendant processes and management undertaken by the students.

2. Acknowledgments

My thanks go to the faculty of the CS department at the College of Charleston for supporting this approach to teaching Software Engineering. In particular, the support of the chair, Dr. Chris Starr, who let me keep working on this project and Dr. Paul Buhler who had the original idea of making this project the basis of a full, required course.

My thanks also to my colleagues on the program committee, all of whom are better than me, to let me participate in this symposium. My particular thanks to Pascal Constanza for his support of the CLforJava project.

And to all the approximately 130 students who have contributed to this project over the last 5 years, my deep gratitude for their hard work and their willingness to become effective teams to build something they didn't think they could do.

3. References

1. Muchnick, S., Advanced Compiler Design & Implementation, Morgan Kaufmann, Academic Press, 1997
2. Boetje, J. Common Lisp for Java, A New Implementation Intertwined with Java, Proceedings of the International Lisp Conference, 2005, Stanford CA.
3. Boetje, J. Unicode 4.0 In Common Lisp, Adoption of Unicode 4.0 in CLforJava, Proceedings of the International Lisp Conference, 2005, Stanford CA.
4. Cotton, J., Boetje, J., A Metaobject Protocol for CLforJava, International Lisp Conference, Cambridge, England, 2007
5. Boetje, J. Foundational Actions: Teaching Software Engineering When Time Is Tight, Proceedings of the Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiSCE), 2006, Bologna, Italy
6. Steele, G. Common Lisp The Language, 2nd Edition, Digital Equipment Corporation 1990
7. Gamma E., Helm R., Johnson R., Vlissides J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995
8. Bloch, J. Effective Java Programming Guide, Sun Microsystems, Inc. Addison-Wesley, 2001
9. Fogel, K., Producing Open Source Software: How to Fun a Successful Free Software Project, O'Reilly Media, <http://producingoss.com/>, 2005



Improving the usability of Kenzo, a Common Lisp system for Algebraic Topology*

Jónathan Heras Vico Pascual Julio Rubio
Francis Sergeraert

{jonathan.heras, vico.pascual, julio.rubio}@unirioja.es,
francis.sergeraert@ujf-grenoble.fr

Abstract

Kenzo is a symbolic computation system devoted to Algebraic Topology. Written in Common Lisp, this program succeeded in computing homology and homotopy groups so far unreachable. The challenge is now to increase the number of users and to improve its usability. Instead of designing simply a friendly front-end, we have undertaken the task of devising a *mediated* access to the system, constraining its functionality, but providing guidance to the user in his navigation on the system. This objective is reached by constructing in Common Lisp an *intermediary layer*, allowing us an *intelligent* access to some features of the system. This intermediary layer is supported by XML technology and interplays between a graphical user interface and the *pure* Kenzo system.

1 Introduction

Kenzo [10] is a Common Lisp system, devoted to Symbolic Computation in Algebraic Topology. It was developed under the direction of the fourth author of this paper, and has been successful, in the sense that it has been capable of computing homology groups unreachable by any other means.

The main features of Kenzo as a Common Lisp system are: (1) the using of the Common Lisp Object System (CLOS) to organize a hierarchy of complex algebraic structures, and (2) the intensive use of higher-order functional programming, allowing us to represent and manipulate *infinite* spaces on a computer. Its power stems from an explicit link between (functional) infinite data structures and some finite counterparts. The first ones are used to encode the complex structures of Algebraic Topology; the second data (as lists, matrices, and the like) are used to compute effectively the invariants associated to the spaces.

Kenzo is in production since 1999. Having detected the accessibility and usability as two weak points in it (implying difficulties in increasing the number of users of the system), several proposals have been studied to interoperate with Kenzo (being the original user interface Common Lisp itself, the search for other

*Partially supported by Comunidad Autónoma de La Rioja, project Colabora2007/16, and Ministerio de Educación y Ciencia, project MTM2006-06513.

ways of interaction seems convenient to extend the use of the system). The aim of this paper is to present a report on our project for giving a new user interface to Kenzo.

Traditionally, symbolic computation systems, and Kenzo is no exception, have been oriented to research. This implies in particular, that development efforts in the area of Computer Algebra systems have been centered in aspects such as the improvement of the efficiency (or the accuracy, in symbolic-numerical systems) or the extension of the scope of the applications. Things are a bit different in the case of widely spread commercial systems such as Mathematica or Maple, where some attention is also payed to connectivity issues or to special-purpose user interfaces (usually related to educational applications). But even in these cases the central focus is on the results of the calculations and not on the interaction with other kind of (software or human) agents.

The situation is, in any sense, similar in the area of interoperability among symbolic computation systems (including here both computer algebra systems and proof assistants). The emphasis has been put in the *universality* of the middleware (see, for instance, [5]). Even if important advances have been achieved, severe problems have appeared, too, such as difficulties in reusing previous proposals and the final obstacle of the speculative existence of a *definitive mathematical interlingua*. The irruption of XML technologies (and, in our context, of MathML [2] and OpenMath [4]) has allowed standard knowledge management, but they are located at the *infrastructure* level, depending always on higher-level abstraction devices to put together different systems. Interestingly enough, the initiative SAGE [21] producing an integrated environment seems to have no use for XML standards, intercommunication being supported by ad-hoc SAGE mechanisms.

In summary, in the symbolic computation area, we are always looking for *more powerful* systems (with more computation capacities or with more general expressiveness). However, it is the case that our systems became so powerful, that we can lose some interesting kinds of users or interactions. We have encountered this situation when designing and developing the *TutorMates project* [13]. TutorMates is aimed at linking an educational front-end with the Maxima system [19]. Since the final users were students (and teachers) at the high school level it was clear from the beginning of the project that Maxima should be *weakened* in any sense, in order to make its outputs meaningful for non mathematics-trained users. This approach is now transferred to the field of symbolic computation in Algebraic Topology, where the Kenzo system [10] provides a complete set of calculation tools, which can be considered difficult to use by a non-Common Lisp trained user (typically, an Algebraic Topology student, teacher or researcher). The key concept is that of *mediated access* by means of an *intermediary layer* aimed at providing an *intelligent middleware* between a user interface and the kernel Kenzo system.

The paper is organized as follows. In the next section a short description of Kenzo as a Common Lisp system is presented. In Section 3 antecedents of our current project are commented, reporting on previous attempts to interoperate with Kenzo and on the TutorMates system. Section 4 gives some insights on methodological and architectural issues, both in the development of the client interface and in the general organization of the software systems involved. The central part of the paper can be found in Section 5, where the basics on the intermediary layer are explained. The concrete state of our project to interface

with Kenzo is the aim of Section 6. The paper ends with two sections devoted to open problems and conclusions, and finally the bibliography.

2 Kenzo as a Common Lisp system

The Kenzo program shows a concrete example of use of CLOS for a relative large implementation work (16000 Common Lisp lines and a 340pp documentation). It is the first significant *machine program* about classical Algebraic Topology. It is not only a program implementing various *known* algorithms; *new* methods have been developed to *transform* the main “tools” of Algebraic Topology, mainly the spectral sequences, not at all *algorithmic* in the traditional organization, into actual *computing* methods. With these “tools” the Kenzo program is able to produce mathematical results that are unreachable otherwise.

2.1 An example of Kenzo work.

Let us show a simple example to illustrate which is possible with this program. The homology group $H_5\Omega^3\text{Moore}(\mathbb{Z}_2, 4)$ ¹ is “in principle” reachable thanks to old methods, see [6], but experience shows even the most skilful topologists meet some difficulties to determine it, see [18, 20]. With the Kenzo program, you construct the Moore space.

```
> (setf m4 (moore 2 4)) ✘
[K1 Simplicial-Set]
```

The program returns the Kenzo-object #1, a simplicial set, that is, a combinatorial version of the Moore space which is asked for, and this object is assigned to the symbol `m4`. Then you construct the third loop-space of this Moore space.

```
> (setf o3m4 (loop-space m4 3)) ✘
[K15 Simplicial-Group]
```

The combinatorial version of the loop space is *highly* infinite: it is a combinatorial version of the space of *continuous* maps $S^3 \rightarrow \text{Moore}(\mathbb{Z}_2, 4)$ but functionally coded as a small set of functions in a `simplicial-group` object, that is, a simplicial set with an added group structure compatible with the simplicial structure. Finally the fifth homology-group is asked for.

```
> (homology o3m4 5) ✘
Homology in dimension 5 :
Component Z/2Z
Component Z/2Z
Component Z/2Z
Component Z/2Z
Component Z/2Z
---done---
```

and the result $H_5\Omega^3\text{Moore}(\mathbb{Z}_2, 4) = \mathbb{Z}_2^5$ is obtained in some seconds in a standard PC. In natural situations a little more complicated, the Kenzo program has already computed new homology groups unreachable so far with “classical” Algebraic Topology, even from a theoretical point of view.

¹The space $\text{Moore}(\mathbb{Z}_2, 4)$ is a “canonical” space having only non-trivial homology in dimension 4, namely \mathbb{Z}_2 , and $\Omega^3\text{Moore}(\mathbb{Z}_2, 4)$, its third loop space, is the space of continuous maps from the 3-sphere S^3 to this Moore space; the challenge is to determine the fifth homology group of this functional space.

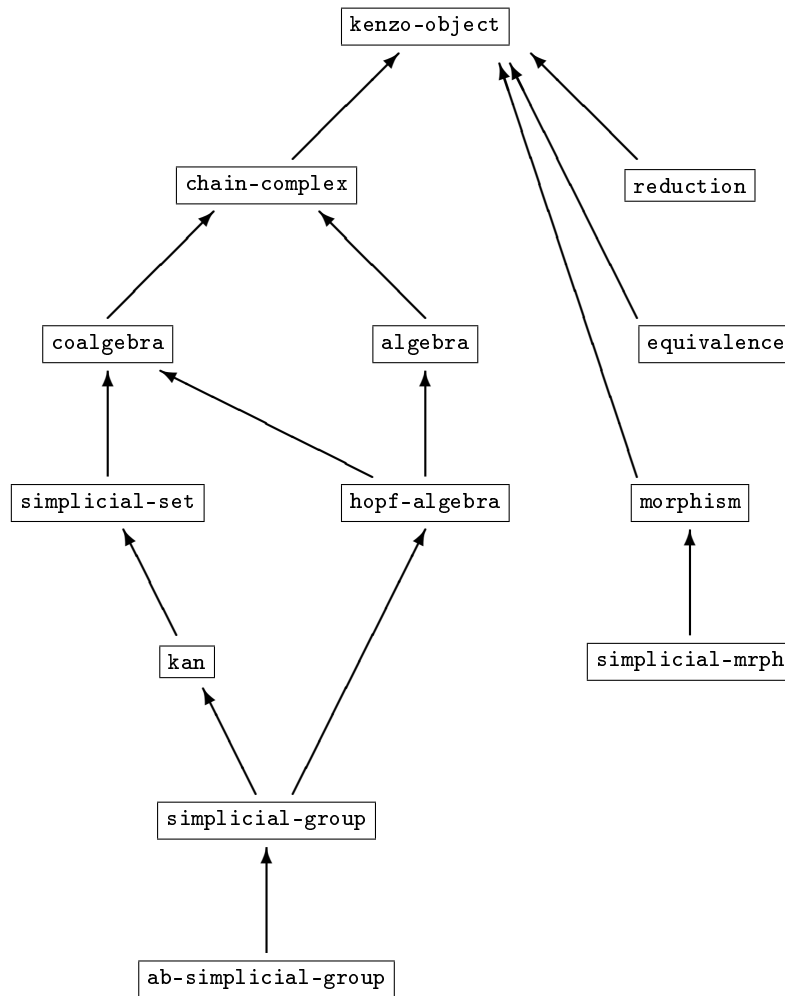


Figure 1: The Kenzo class diagram.

2.2 Kenzo classes.

Figure 1 shows the class diagram of Kenzo objects. The lefthand part of the class diagram is made of the main mathematical categories that are used in combinatorial Algebraic Topology. A *chain complex* is a graded differential module; an *algebra* is a chain complex with a compatible multiplicative structure, the same for a *coalgebra* but with a comultiplicative² structure. If a multiplicative and a comultiplicative structures are added and if they are compatible with each other in a natural sense, then it is a *Hopf algebra*, and so on.

The *hopf-algebra* and *simplicial-group* classes are typical cases where a *multi-heritage* situation is met; we show the *actual* Kenzo definitions of these classes.

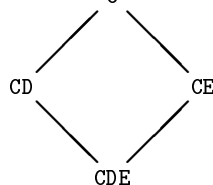
²That is, some *cooperator* $A \rightarrow A \otimes A$.

```
.....
(DEFCLASS HOPF-ALGEBRA (coalgebra algebra)
  ())
```

```
(DEFCLASS SIMPLICIAL-GROUP (kan hopf-algebra)
  ((grml :type simplicial-mrph :reader grml1)
   (grin :type simplicial-mrph :reader grin1)))
.....
```

You see the definition of the `hopf-algebra` class is particularly striking; it explains that a Hopf-algebra is nothing but an algebra *and* a coalgebra; the compatibility conditions between both structures *cannot* be verified by a program and they necessarily depend on the programmer's "lucidity". In the same way, a *simplicial group* is a `kan` object and a `hopf-algebra` object sharing some common data, namely a coalgebra structure, with two further slots, `grml` (group multiplication) and `grin` (group inversion), those slots being some simplicial morphisms.

In such a multi-heritage situation, it is important the `call-next-method` function works as hoped-for. Look at this artificial situation just to show the process; the `C` class has two subclasses `CD` and `CE`, which have in common the subclass `CDE`; the artificial `initialize-instance` methods let you verify that `call-next-method` remembers its story when deciding what really the *next method* must be. Here, when processing the `CD`-level, `call-next-method` "remembers" the process was initiated from the `CDE`-level, so that the `CE`-level stage is not forgotten.



```
.....
> (defclass C () ()) ✘
#<STANDARD-CLASS C>
> (defclass CD (C) ()) ✘
#<STANDARD-CLASS CD>
> (defclass CE (C) ()) ✘
#<STANDARD-CLASS CE>
> (defclass CDE (CD CE) ()) ✘
#<STANDARD-CLASS CDE>
> (defmethod initialize-instance ((c c) &rest rest)
  (print "C-initialization")) ✘
#<STANDARD-METHOD INITIALIZE-INSTANCE (C)>
> (defmethod initialize-instance ((cd cd) &rest rest)
  (print "beginning CD-initialization")
  (call-next-method)
  (print "finishing CD-initialization")) ✘
#<STANDARD-METHOD INITIALIZE-INSTANCE (CD)>
> (defmethod initialize-instance ((ce ce) &rest rest)
  (print "beginning CE-initialization")
  (call-next-method)
  (print "finishing CE-initialization")) ✘
#<STANDARD-METHOD INITIALIZE-INSTANCE (CE)>
> (defmethod initialize-instance ((cde cde) &rest rest)
  (print "beginning CDE-initialization")
  (call-next-method)
  (print "finishing CDE-initialization")) ✘
#<STANDARD-METHOD INITIALIZE-INSTANCE (CDE)>
```

```

> (make-instance 'C) ✘
"C-initialization"
#<C @ #x212184da>
> (make-instance 'CD) ✘
"beginning CD-initialization"
"C-initialization"
"finishing CD-initialization"
#<CD @ #x21220e8a>
> (make-instance 'CE) ✘
"beginning CE-initialization"
"C-initialization"
"finishing CE-initialization"
#<CE @ #x2122698a>
> (make-instance 'CDE) ✘
"beginning CDE-initialization"
"beginning CD-initialization"
"beginning CE-initialization" ←←←!!!
"C-initialization"
"finishing CE-initialization"
"finishing CD-initialization" ←←←!!!
"finishing CDE-initialization"
#<CDE @ #x2122c03a>

```

And you may also play with the *auxiliary* `:before`, `:after` and `:around` methods to order as you like the various initialization steps. As a typical example, when the essential part of the initialization work of any `kenzo-object` is done, then the object is *finally* pushed in a list which is used later as explained in the next section. This is obtained as follows.

```

(DEFMETHOD INITIALIZE-INSTANCE :after ((k kenzo-object) &rest rest)
  (push k *k-list*))

```

In this way this is done if and only if the initialization work is successfully finished, even for the more specialized structures: if for example the specialized initialization work for a simplicial set fails and stops on error, then the pushing statement concerning the weakest structure is not run.

2.3 Optimizing computations.

The Kenzo program is certainly a *functional* system. It is frequent that several thousands of functions are present in memory, each one being *dynamically* defined from other ones, which in turn are defined from other ones, and so on. In this quite original situation, the same calculations are frequently *asked again*. To avoid repeating these calculations, it is better to store the results and to systematically examine for each calculation whether the result is already available (*memoization* strategy).

Because of this situation, it is very important not to have *several copies* of the same function; otherwise it is impossible for one copy to guess some calculation has already been done by another copy. This is a very important question in this program, so that the following idea has been used. Each Kenzo object has a rigorous *definition*, stored as a list in the `orgn` slot of the object (`orgn` stands for *origin* of the object). This is the main reason of the top class `kenzo-object`: making easier this process. The actual definition of the `kenzo-object` class:


```

.....
(DEFCLASS KENZO-OBJECT ()
  ((idnm :type fixnum :reader idnm)
   (orgn :type list :reader orgn)
   (prpr :type list :reader prpr)
   (cmmn :type list :reader cmmn)))
.....

```

Then, when any `kenzo-object` is to be considered, its *definition* is constructed and the program firstly looks in `*k-list*` whether some object corresponding to this definition already exists; if yes, no `kenzo-object` is constructed, the already existing one is simply returned. Look at this small example where we construct the second loop space of S^3 , then the first loop space, and then again the second loop space. In fact the initial construction of the second loop space required the first loop space, and examining the identification number `K??` of these objects shows that when the first loop space is later asked for, Kenzo is able to return the already existing one.

```

.....
> (setf s3 (sphere 3)) ✘
[K372 Simplicial-Set]
> (setf o2s3 (loop-space s3 2)) ✘
[K380 Simplicial-Group]
> (setf os3 (loop-space s3 1)) ✘
[K374 Simplicial-Group]
> (setf o2s3-2 (loop-space s3 2)) ✘
[K380 Simplicial-Group]
> (eq o2s3 o2s3-2) ✘
.....

```

T.

The last statement shows the symbols `o2s3` and `o2s3-2` points to the same machine address. In this way we are sure any `kenzo-object` has no duplicate, so that the memory process for the values of numerous functions cannot miss an already computed result. Let us look some `orgn` slots:

```

.....
> (orgn o2s3) ✘
(LOOP-SPACE [K374 Simplicial-Group])
> (orgn (k 374)) ✘
(LOOP-SPACE [K372 Simplicial-Set])
> (orgn (k 372)) ✘
(SPHERE 3)
.....

```

You see in this way the history of the construction process can be freely examined by the user, which is important in the development stage.

2.4 Delaying initializations.

The complete structure of a Kenzo object is extremely complicated, and many components are often useless. Another CLOS feature is therefore used to avoid the maybe non-necessary initialization works. The following artificial example explains how this is possible; it is a kind of *autoloading* mechanism, elegant, easy to be used, and useful to avoid initializing needless slots. We assume a `F` class, where each `F` object has two slots, `s11` and `s12`; the first one is necessary, but the second one would be the result of a *complex* process here simulated as being 1000 times the value of the first one.

```

.....
> (DEFCLASS F ()
  ((s11 :type integer :initarg :s11 :reader s11)
   (s12 :type integer :reader s12))) ✘
#<STANDARD-CLASS F>
.....

```

```

> (DEFMETHOD SLOT-UNBOUND (class (fi f) (slot-name (eql 's12)))
  (declare (ignore class))
  (setf (slot-value fi 's12) (* 1000 (s11 fi)))
  (s12 fi)) ✘
#<STANDARD-METHOD SLOT-UNBOUND (T F (EQL SL2))>
> (SETF FI (make-instance 'f :s11 23)) ✘
#<F @ #x213a7b8a>
> (SLOT-BOUNDP fi 's12) ✘
NIL
> (s12 fi) ✘
23000
> (SLOT-BOUNDP fi 's12) ✘
T

```

You see the generic function `slot-unbound` is available which is called by the error manager when a non-initialized slot is asked for. The standard process finally does generate an error. But the user can write specialized methods for this generic function, allowing him instead to initialize the missing slot by some process using the available information. You see the initialization process lets uninitialized the `s12` slot of the `F`-instance located by `fi`, but when this slot is asked for, the “right” value is in fact returned! A new examination by `slot-boundp` shows the slot is now bound.

This process is extremely convenient to organize the data as a living object where each time some missing component is questioned, an automatic “repairing process” is started, computing the missing information. The process may be recursive, so that if, in the repairing process, some other datum is again missing, an other repairing process is recursively started, and so on.

This possibility is intensively used in the Kenzo program. Look at this small experience. Firstly we reinitialize the environment by `cat-init`. When the fourth loop space $\Omega^4 S^5$ is constructed, you see only 26 Kenzo objects are present in the environment. Then the homology group $H_2 \Omega^4 S^5$ is asked for. The answer, \mathbb{Z}_2 is quickly obtained, but the number of present Kenzo objects is now 504; an enormous set of `slot-unbound` calls has generated the construction of 478 new Kenzo objects, necessary to do the calculation. Furthermore a `:before` method had been added just to count the number of `slot-unbound` calls, a convenient debugging trick; you see the homology calculation has recursively generated 240 `slot-unbound` calls.

```

> (cat-init) ✘
---done---
> (setf s5 (sphere 5)) ✘
[K1 Simplicial-Set]
> (setf o4s5 (loop-space s5 4)) ✘
[K21 Simplicial-Group]
> (length *k-list*) ✘
26
> (setf counter 0) ✘
0
> (defmethod slot-unbound :before (class instance slot)
  (declare (ignore class instance slot))
  (incf counter)) ✘
#<STANDARD-METHOD SLOT-UNBOUND :BEFORE (T T T)>
> (homology o4s5 2) ✘
Homology in dimension 2 :
Component Z/2Z
---done---

```

```

> (length *k-list*) ✘
504
> counter ✘
240

```

2.5 Mixing low level and high level programming.

Computing time is crucial for the applications of the Kenzo program. The complexity of the implemented algorithms is highly exponential, so that the developer must carefully consider how he can improve the computing time of the written down Lisp code. In particular, if the heart of the program may be written close to the machine language, large amounts of computing time can be saved. But conversely this must not penalize the *readability* and the *modularity* of the program.

Which is striking with Common Lisp is the possibility of easily mixing *low level* and *high level* programming. The features about OOP show how Common Lisp is powerful in high level programming, allowing the user to directly handle the sophisticated objects of Algebraic Topology such as chain complexes, products and coproducts, Hopf algebras, simplicial sets and simplicial groups.

But on the other hand, the Kenzo program intensively uses the low level part of the Common Lisp language, that is, the quasi-assembler language which is the very root of the language, such as the popular `car`, `cdr`, and `cons`. This is possible thanks to the Common Lisp *macrogenerator*. Let us consider the case of the type `absm`, that is, *abstract simplex*. These objects are really the most elementary constituents of the Kenzo geometric objects, and they are so intensively used, billions of times for every significant Kenzo run, that you *must not* use CLOS for these kernel structures. Kenzo defines the `absm` type as follows:

```

(DEFUN ABSM-P (object)
  (declare (type any object))
  (the boolean
    (and (consp object)
          (eq :absm (car object))
          (typep (cdr object) 'iabsm))))

(DEFTYPE ABSM () '(satisfies absm-p))

```

The `absm-p` function explains an `absm` is a `cons` (pair) where the lefthand component is the keyword `:absm` and the righthand one is an `iabsm`, that is, an *internal absm*; in the same way, elsewhere in the program, it is explained an `iabsm` is again a `cons` where the righthand component is anything and the lefthand component is a fixnum coding a *degeneracy operator*. Most of computations in Algebraic Topology are in fact low level computations about degeneracy operators where such an operator is a decreasing list of small integers, like (5 2 0); because this list is *strictly* decreasing, it can be represented by the fixnum 37 because $37 = 2^5 + 2^2 + 2^0$, so that all the standard calculations about degeneracy operators become fine calculations *at the bit level* on binary fixnums. But Common Lisp has all the predefined functions to do such a job, so that the programmer can efficiently work according to this strategy. A considerable memory space is saved so and furthermore the calculations are much faster.

If a degeneracy operator is to be extracted from an `absm`, the `dgop` macro is used:

```

.....
> (DEFMACRO DGOP (absm)
  '(the dgop (cadr (the cadr ,absm))) ✘
DGOP
> (macroexpand '(dgop argument)) ✘
(THE DGOP (CADR (THE ABSM ARGUMENT)))
.....
which explains that in fact the call of dgop is synonymous with a call of the
assembler-like cadr, but the types of argument and result are verified:
.....
> (dgop (absm 37 'something)) ✘
37
> (dgop 'not-an-absm) ✘
Error: object "NOT-AN-ABSM" is not of type "ABSM".
[condition type: PROGRAM-ERROR]
.....

```

When the program is compiled, the compiler firstly translates the source code when a macro call is found, so that it is an assembler-like statement which is compiled; furthermore an appropriate compiler option allows the compiled code to ignore or not the type verifications through the `'the'` statements. When the program is finalized for production work, of course these type verifications are discarded to save computing time. You see in this way the Lisp code is *readable*, this code being firstly translated in low level Lisp statements, therefore very efficiently compiled, without loosing if necessary the type verifications.

3 Antecedents of our project

As explained in the Introduction, several proposals have been studied to inter-operate with Kenzo. The most elaborated approach was reported in [1]. There, we devised a remote access to Kenzo, using CORBA [17] technology. An XML extension of MathML played a role there too, but just to give genericity to the connection (avoiding the definition in the CORBA Interface Description Language [17] of a different specification for each Kenzo class and datatype). There was no intention of taking profit from the semantics possibilities of MathML. Being useful, this approach ended in a prototype, and its enhancement and maintenance were difficult, due both to the low level characteristics of CORBA and to the pretentious aspiration of providing *full* access to Kenzo functionalities. We could classify the work of [1] in the same line as [5] or the initiative IAMC [15], where the emphasis is put into powerful and generic access to symbolic computation engines.

On the contrary, the TutorMates project [13] had, from its very beginning, a much more modest objective. The idea was to give access just to a part of Maxima, but guiding the user in his interaction. Since the purpose of TutorMates was educational (high school level), it was clear that many outputs given by Maxima were unsuitable for the final users, depending on the degree and the topic learned in each TutorMates session. To give just an example, an imaginary solution to a quadratic equation has meaning only in certain courses. In this way, a *mediated* access to Maxima was designed. The central concept is an intermediary layer that communicates, by means of an extension of XML, between the graphical user interface (Java based) and Maxima. The extension of MathML allows us to encode a *profile* for the interaction. A profile is composed of a role (student or teacher), a level and a lesson. In the case of a teacher (supposed to be preparing material for his students), full access to Maxima outputs is given, but a *warning* indicates to him whether the answer would be suitable

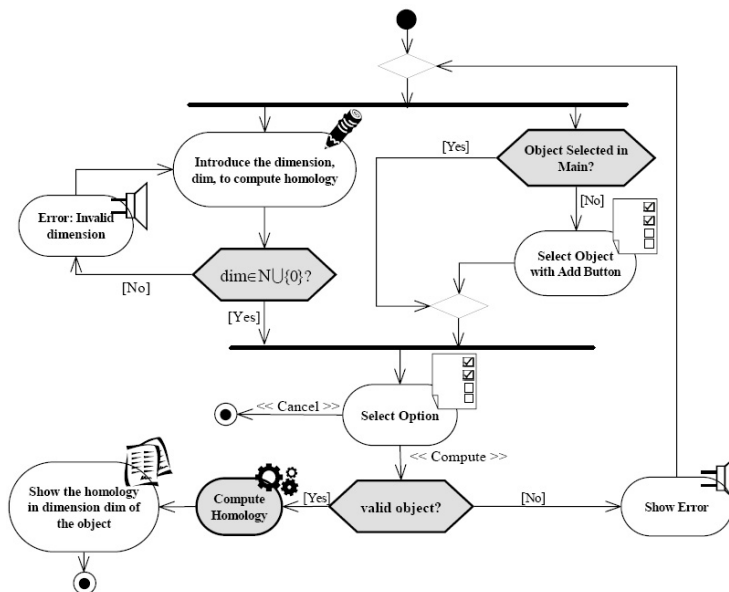


Figure 2: A fragment of the control and navigation graph.

inside the level and the lesson encoded in the profile. In this way, the intermediary layer allows the programmer to get an *intelligent* interaction, different from the “dummy” remote access obtained in [1].

Now, our objective is to emulate this TutorMates organization in the Kenzo context. The final users could be researchers in Algebraic Topology or students of this discipline. The problems to be tackled in the intermediary layer are different from those of TutorMates. The methodological and architectural aspects of this new product are presented in the following section.

4 Methodological and Architectural Issues

We have tried to guide our development with already proven methodologies and patterns. In the case of the design of the interaction with the user in our GUI front-end³, we have followed the guidelines of the Noesis method [7]. In particular, our development has been supported by some Noesis models for control and navigation in user interfaces (see an example in Figure 2).

Even if graphical specification mechanisms have well-known problems (related with their scalability), Noesis models provide *modular tools*, allowing the designer to control the complexity due to the size of graphics. These models enable an exhaustive traversal of the interfaces, detecting errors, disconnected areas, lack of homogeneity, etc.

With respect to the general organization of the software system, we have been inspired by the *Microkernel* architectural pattern [3]. This pattern gives a global view as a *platform*, in terminology of [3], which implements a virtual

³The GUI has been implemented using the package *Common Graphics* and the *Integrated Development Environment* of Allegro Common Lisp [11].

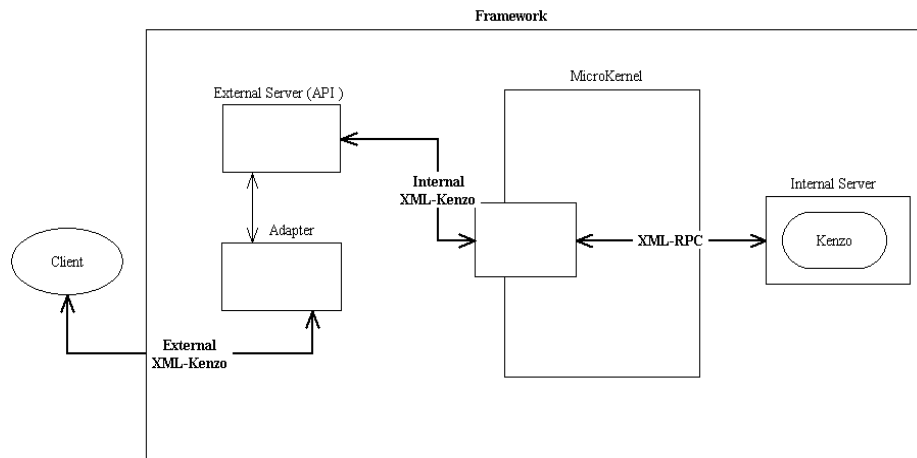


Figure 3: Microkernel architecture of the system.

machine with applications running on top of it, namely a *framework* (in the same terminology). A high level perspective of the system as a whole is shown in Figure 3. Kenzo itself, wrapped with an interface based on XML-RPC [22], is acting as *internal server*. The *microkernel* acting as intermediary layer is based on an XML processor, allowing both a link with the standard XML-RPC used by Allegro Common Lisp [11], and intelligent processing. The view of the *external server* is again based on an XML processor, with a higher level of abstraction (since mathematical knowledge is included there) which can map expressions from and to the microkernel, and which is decorated with an *adapter* (the *Proxy* pattern, [12], is used to implement the adapter), establishing the final connection with the client, a Graphical User Interface in our case. A simplified version of the Microkernel pattern (without the external server) would suffice if our objective was to build a GUI for Kenzo. But we also pursue extending Kenzo by wrapping it in a framework which will link any possible client (other GUIs, web applications, web services, ...) with the Kenzo system. In this sense, our GUI is a client of our framework. The framework should provide each client with all necessary mathematical knowledge.

Which aspects of the intelligent processing must be dealt with in the external server or in the microkernel, is still controversial (in the current version, as we will explain later, we have managed the questions related to the input specifications in the external server and the most important mediations are done at the microkernel level). Moreover, the convenience of a double level of processing is clear, being based on, at least, two reasons. On the one hand the more concrete one (microkernel) is to be linked to Kenzo (via XML-RPC) and the more abstract one is aimed at being exported and imported, rendered by (extended) MathML engines, and so on. On the other hand, this double level of abstraction reflects the different languages in which the knowledge has to be expressed. The external one is near to Algebraic Topology, and it should offer a communication based on the concepts of this discipline to the final clients (this gives a small type system; see Section 5). The internal part must communicate with Kenzo, and therefore a low level register of each session must be maintained (for instance, the unique identifier referring to each object, in order to avoid recalculations).

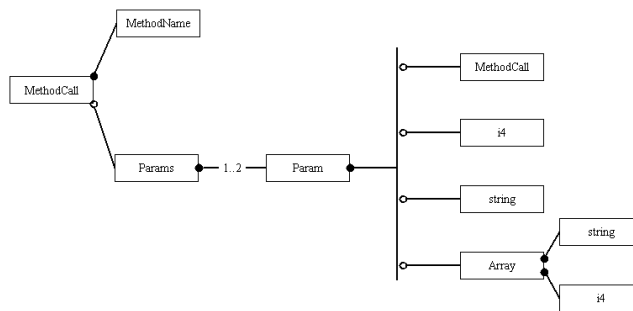


Figure 4: Description of the Internal XML Kenzo Schema.

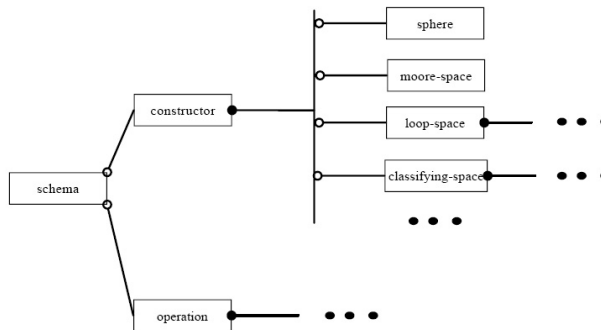


Figure 5: Fragment of the External XML Kenzo Schema.

There, a procedural language based on Kenzo conventions is needed.

As explained before, XML gives us the universal tool to transmit information along the different layers of the system. Besides the XML-RPC mechanism used by Allegro Common Lisp, two more XML formats (defined by means of XML schemas) are to be considered. The first one (used in the microkernel) is diagrammatically described in Figure 4, by using the Noesis method [9] again. The second format, used in the external server, will be (it is not completely defined yet) presented as an extension of the MathML schema [2]. Figure 5 shows a diagram corresponding to a part of this schema. The structure of this XML schema allows us to represent some knowledge on the process (for instance, it differentiates constructors from other kinds of algebraic manipulations); other more complex mathematical knowledge can not be represented in the syntax of the schema (see Section 5). In Figure 6, we show how a Kenzo command (namely, the calculation of the third group of homology of the sphere of dimension 3) will be transformed from the user command on the GUI (top part of the figure) to the final XML-RPC format (the conventional Lisp call is shown, too; however our internal server, Kenzo wrapped with an XML-RPC interface, will execute the command directly).

In the next section the behavior pursued with this architecture is explained.

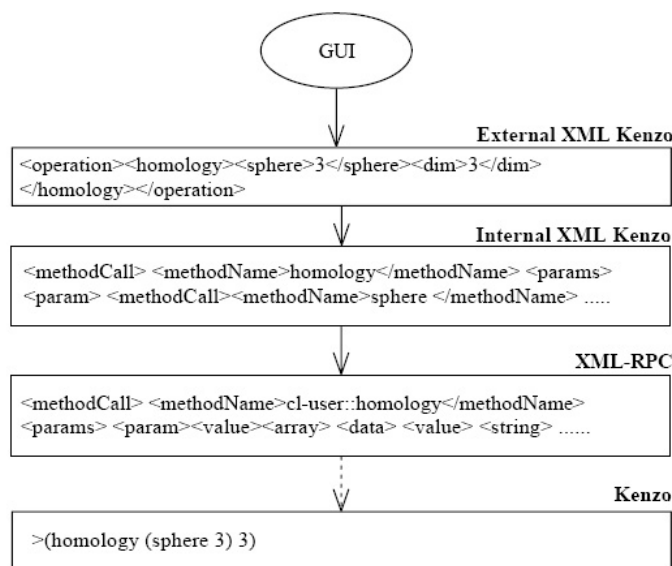


Figure 6: Transforming XML representations.

5 Knowledge Management in the Intermediary Layer

The system as a whole will improve Kenzo including the following “intelligent” enhancements:

1. Controlling the input specifications on constructors.
2. Avoiding some operations on objects which will raise errors.
3. Chaining methods in order to provide the user with new tools.
4. Determining if a calculation can be done in a local computer or should be derived to a remote server.

The first aspect is attained, in an integrated manner, inside the Graphical User Interface. The three last ones are dealt with in the intermediary layer. From another point of view, the first three items are already partially programmed in the current version of the system; the last one is further work.

In order to explain the differences between points 1 and 2, it is worth noting that in Kenzo there are two *kinds* of data. The first one is representing *spaces* in Algebraic Topology (by *spaces* we mean here, any data structure having both behavior and elements belonging to it, such as a simplicial set, a simplicial group, a chain complex, and so on). The second kind of data is used to represent *elements* of the spaces. Thus, in a typical session with Kenzo, the users proceed in two steps: first, constructing some spaces, and second, applying some operators on the (elements of the) spaces previously built. This organization in two steps has been described by using Algebraic Specification methods in [16] and [8], for instance. Therefore, the first item in the enumeration refers to the inputs for

the constructors of spaces, and the second item refers to some operations on *concrete* spaces. As we are going to explain, the first kind of control is naturally achieved in the GUI client (from the mathematical knowledge provided by the external XML format) but the second one, which needs some expert knowledge management, is better dealt with in the intermediary layer.

Kenzo is, in its pure mode, an untyped system (or rather, a dynamically typed system), inheriting its power and its weakness from Common Lisp. Thus, for instance, in Kenzo a user could apply a constructor to an object without satisfying its input specification. For example, the method constructing the classifying space of a simplicial group could be called on a simplicial set without a group structure over it. Then, at runtime, Common Lisp would raise an error informing the user of this restriction. This is shown in the following fragment of a Kenzo session:

```

> (loop-space (sphere 4)) ✘
[K6 Simplicial-Group]
> (classifying-space (loop-space (sphere 4))) ✘
[K18 Simplicial-Set]
> (sphere 4) ✘
[K1 Simplicial-Set]
> (classifying-space (sphere 4)) ✘
;; Error: No method in generic function CLASSIFYING-SPACE
;; is applicable to arguments: [K1 Simplicial-Set]

```

With the first command, namely `(loop-space (sphere 4))`, we construct a simplicial group. Then, in the next step we are verifying that a simplicial group has a classifying space (which is, in general, just a simplicial set). In the third command, we check that the sphere of dimension 4 is constructed in Kenzo as a simplicial set. Thus, when in the last command we try to construct the classifying space of a simplicial set, the Common Lisp Object System (CLOS) raises an error.

In the current version of our system this kind of error is controlled, because the inputs for the operations between spaces can be only selected among the spaces with suitable characteristics. The equivalent in our system of the example introduced before in pure Kenzo, is shown in Figure 7, where it can be seen that for the classifying operation just the spaces which are simplicial groups are candidates to be selected. This enriches Kenzo with a small (semantical) type system which will be defined into the external XML schema.

With respect to the second item in the previous enumeration, the most important example in the current version is the management of the *connection degree* of spaces. Kenzo allows the user to construct, for instance, the loop space of a non simply connected space (as the sphere of dimension 1). The result is a simplicial set on which some operations (for instance, to compute the set of faces of a simplex) can be achieved without any problems. On the contrary, theoretical results ensure that the homology groups are not of finite type, and then they cannot be computed. In pure Kenzo, the user could ask for a homology group of such an space, catching a runtime error.

In our current version of the system, the intermediary layer includes a small expert system, computing, in a symbolic way (that is to say, working with the *description* of the spaces, and not with the spaces themselves considered as Common Lisp objects), the connection degree of a space. The set of rules gives a connection degree to each space builder (for instance, a sphere of dimension n has connection degree $n - 1$), and then a rule for each operation on spaces. For

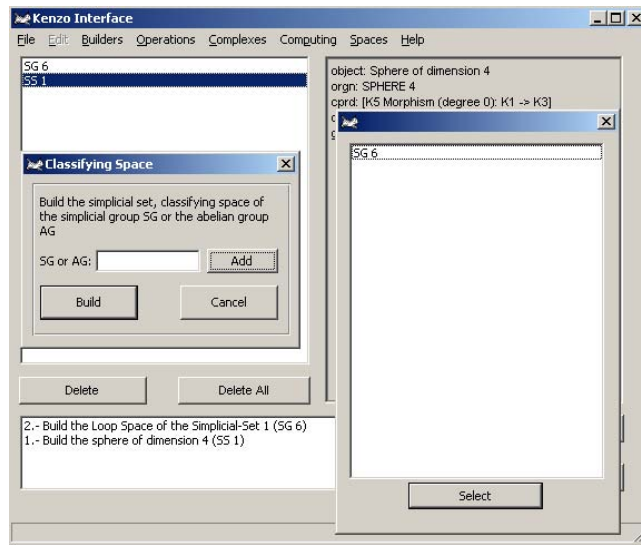


Figure 7: Screen-shot of Kenzo Interface with a session related to classifying spaces.

instance, loop space decreases the connection degree of its input in one unity, suspension increases it in one unity, a cartesian product has, as connection degree, the minimum of the connection degrees of its factors, and so on. From the design point of view, a *Decorator* pattern [12] was used, decorating each space with an annotation of its connection degree in the intermediary layer. Then, when a computation (of a homology group, for instance) is demanded by a user, the intermediary layer monitors if the connection degree allows the transferring of the command to the Kenzo kernel, or a warning must be sent through the external server to the user.

As for item three, the best example is that of the computation of *homotopy groups*. In pure Kenzo, there is no final function allowing the user to compute them. Instead, there is a number of complex algorithms, allowing a user to chain them to get some homotopy groups. Our current user interface has an option to compute homotopy groups. The intermediary layer is in charge of chaining the different algorithms present in Kenzo to reach the final objective. In addition, Kenzo, in its current version, has limited capabilities to compute homotopy groups (depending on the homology of Eilenberg-Mac Lane spaces that are only partially implemented in Kenzo), so the *chaining* of algorithms cannot be *universal* (in this case, a possibility would be to *wire* the enhancement in the GUI, by means of the external XML schema, as in the case of item 1). Thus, the intermediary layer should process the call for a homotopy group, making some consultations to the Kenzo kernel (computing some intermediary homology groups, for instance) before deciding if the computation is possible or not (this is still work in progress).

Regarding point four, our system can be distributed, at present, in two manners: (a) as a stand-alone application, with a heavy client containing the Kenzo kernel to be run in the local host computer; (b) as a light client, containing just the user interface, and every operation and computation is done in a re-

remote server (with the *AllegroServe* technology). The second mode has obvious drawbacks related to the reliability of Internet connections, to the overhead of management where several concurrent users are allowed, etc. But option (a) is not fully satisfactory since interesting Kenzo computations used to be very time and space consuming (requiring, typically, several days of CPU time on powerful computing servers). Thus a mixed strategy should be convenient: the intermediary layer should decide if a concrete calculation can be done in the local computer or it deserves to be sent to a specialized remote server. (In this second case, as it is not sensible to maintain open an Internet connection for several days waiting for the end of a computation, some reactive mechanism should be implemented, allowing the client to disconnect and to be subscribed in some way, to the process of computation in the remote server). The difficulties of this point have two sources: (1) the knowledge here is not based on well-known theorems (as was the case in our discussion on the *connection degree* in the second item of the enumeration), since it is context-dependent (for instance, it depends on the computational power of a local computer), and so it should be based on *heuristics*; (2) the technical problems to obtain an optimal performance are complicated, due, in particular, to the necessity of maintaining a *shared state* between two different computers. These technical aspects are briefly commented in the Open Problems section.

With respect to the kind of heuristic knowledge to be managed into the intermediary level, there is some part of it that could be considered obvious: for instance, to ask for an homology group $H_n(X)$ where the degree n is big, should be considered harder than if n is small, and then one could wonder about a limit for n before sending the computation to a remote server. Nevertheless, this simplistic view is to be moderated by some expert knowledge: it is the case that in some kinds of spaces, difficulties decrease when the degree increases. The heuristics should consider each operation individually. For instance, it is true that in the computation of homology groups of iterated loop spaces, difficulties increase with the degree of iteration. Another measure of complexity is related to the number of times a computation needs to call the Eilenberg-Zilber algorithm (see [10]), where a double exponential complexity bound is reached. Further research is needed to exploit the expert knowledge in the area suitably, in order to devise a systematic heuristic approach to this problem.

6 State of the Project

The work done up to now has allowed us to reach one of the objectives: code reuse. This reusing has two aspects:

1. We have left the Kenzo kernel untouched. This was a goal since the team developing the framework and the user interface, and the team maintaining and extending Kenzo are different. Therefore, it is convenient to keep both systems as uncoupled as possible.
2. The intermediary level has been used, without changes, both in the stand-alone local version and in the light client with remote server version. A first partial prototype, moving the view towards a web application client (by using *AllegroWebActions*), seems to confirm that the degree of abstraction and genericity reached in our architecture (note that our framework

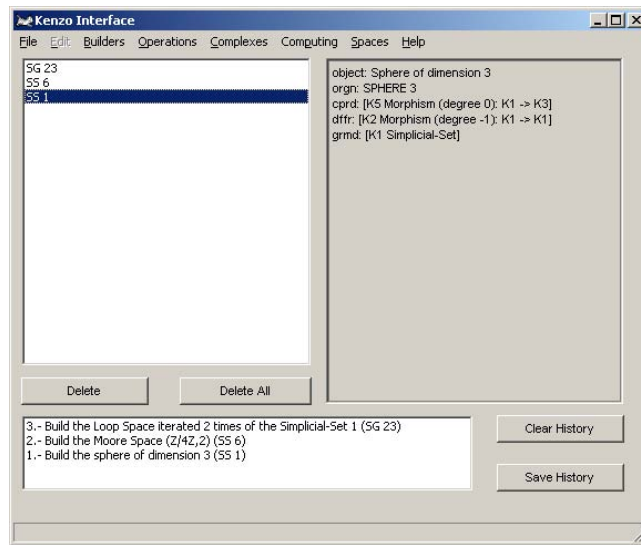


Figure 8: Screen-shot of Kenzo Interface with an example of session.

including several XML formats, each one with different abstraction level) is suitable.

In Figure 8, a screen-shot of our GUI is presented. The main toolbar is organized into 8 menus: *File*, *Edit*, *Builders*, *Operations*, *Complexes*, *Computing*, *Spaces* and *Help*. The rest of the screen is separated into three areas. On the left side, a list with the spaces already constructed during the current session is maintained. When a space is selected (the one denoted by *SS 1* in Figure 8), a description of it is displayed in the right area. At the bottom of the screen, one finds a *history* description of the current session, which can be cleared or saved into a file. It is important to understand that a *history file* is different from a *session file*. The first one is just a plain text description of the commands selected by the user. The second kind of files is described in the next paragraph.

In the current version the *File* menu has just three options: *Exit*, *Save Session* and *Load Session*. When saving a session, a file is produced containing an XML description of the commands executed by the user in that session. In Figure 9 an example of session file can be found, together with a correspondence with their Kenzo counter-parts. At this time, these session files are stored using the standard XML-RPC but our goal, as we show in Figure 9, is to use the external XML schema described in Section 4 (see Figure 5). In this way the session files will be exportable (to be rendered in standard displays, for instance) and even editable from different applications.

The constructors of the spaces we have referred to the first point of Section 5, are collected by the menus *Builders*, *Operations* and *Complexes*. More specifically, the menu *Builders* includes the main ways of constructing new spaces from scratch in Kenzo as options: spheres, Moore spaces, Eilenberg-Mac Lane spaces, and so on. The menu *Operations* refers to the ways where Kenzo allows the construction of new simplicial spaces from other ones: loop spaces, classifying spaces, Cartesian products, suspensions, etc. The menu *Complexes* is

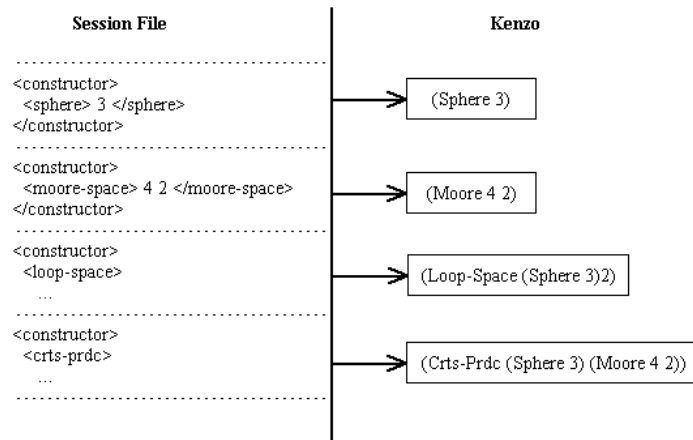


Figure 9: Sample of a session file.

similar, but related to chain complexes instead of simplicial objects (here, for instance, the natural product is the tensorial product instead of the cartesian one).

The menus *Computing* and *Spaces* collect all the operations on concrete spaces (instead of *constructing spaces*, as in the previous cases). Both of them provide their items with all the necessary “intelligence” in order to avoid raising runtime errors. In *Computing* we concentrate on calculations *over* a space. We offer to compute homology groups, to compute the same but with explicit generators and to compute homotopy groups, in this last case we find the third kind of enhancement. In menu *Spaces* currently we only offer the possibility of showing the structure of a simplicial object (this is only applicable to *effective*, finite type spaces).

To consider a first complete (beta) version of the system, it is necessary to complete the questions already mentioned in the text relating to finishing the external XML schema definition and to controlling the cases in which homotopy groups can be effectively computed by Kenzo.

Moreover, we have planned to develop two more tools:

1. In the menu *Builders*, there is a still inactivated slot called *Build-finite-ss*, aimed at emulating, in our environment, the utility present in pure Kenzo which allows the user to construct step-by-step, in an interactive manner, a finite simplicial set (checking, in each step, whether faces are glued together in a coherent way). To this aim, we are thinking of designing a graphical tool.
2. In the menu *Spaces*, it is necessary to include the possibility of operating locally *inside* a selected space. For instance, given a simplex to compute one of its faces or given two simplexes in the same dimension we can compute its product in a selected simplicial group. The difficulty here is related to designing an editor for elements (data of the second kind, using the terminology in Section 5), which can be given as inputs to the local operations. This will give content to the *Edit* menu, in the main toolbar, which is now inactivated.

These extra functionalities are rather a matter of standard programming, and it is foreseen that no research problem will appear when tackling them. The questions discussed in the next section, on the contrary, could imply important challenges.

7 Open Problems

The most important issue to be tackled in the next versions of the system is how organizing the decision on when (and how) a calculation should be derived to a remote server. To understand the nature of the problem it is necessary to consider that there are two kinds of *state* in our context. Starting from the most simple, the *state of a session* can be described by means of the spaces that have been constructed so far. Then, to encode (and recover) such a state, a session file as explained in the previous section would be enough: an XML document containing a sequence of calls to different constructors and methods. In this case, when a calculation is considered too hard to be computed in a local computer, the whole session file could be transmitted to the remote server. There, executing step-by-step the session file, the program will re-find the same state of the local session, proceeding to compute the desired result and sending it to the client. Of course, as mentioned previously, some kind of subscription tool should be enabled, in such a way that the client could stop its running, and then to receive the result (or a notification indicating the result is already available somewhere), after some time (perhaps some days or weeks of computation on the remote server).

Even if this approach can be considered reasonable as a first step, it has turned out to be too simplistic to deal with the richness of Kenzo. A space in Kenzo consists in a number of methods describing its behavior (explaining, for instance, how to compute the faces of its elements). Due to the high complexity of the algorithms involved in Kenzo, a strategy of *memoization* has been systematically implemented, as we already commented in Section 2.3. As a consequence, the *state of a space* evolves after it has been used in a computation (of a homology group, for instance). Thus, the time needed to compute, let us say, a face, depends on the concrete states of every space involved in the calculation (in the more explicit case, to re-calculate a face on a space could be negligible in time, even if in the first occasion this was very time consuming). This notion of *state of a space* is transmitted to the notion of *state of a session*. We could speak of two *states of a session*: the one *shallow* evoked before, that is essentially *static* and can be recovered by simply re-executing the top-level constructor calls; and the other *deep state* which is dynamic and depends on the computations performed on the spaces.

To analyse the consequences of this Kenzo organization, we should play with some scenarios. Imagine during a local session a very time consuming calculation appears; then we could simply send the *shallow state of the session* to the remote server, because even if some intermediary calculations have been stored in local memory, they can be re-computed in the remote server (finally, if they are cheap enough to be computed on the local computer, the price of re-computing them in the powerful remote server would be low). Once the calculation is remotely finished, there is no possibility of sending back the *deep state* of the remote session to the local computer because, usually, the memory

used will exhaust the space in the local computer. Thus, it could seem that to transmit the *sallow state* would be enough. But, in this picture, we are losing the very reason why Kenzo uses the memoization (dynamic programming) style. Indeed, if after obtaining a difficult result (by means of the remote server) we resume the local session and ask for another related difficult calculation, then the remote server will initialize a new session from scratch, being obligated to recalculate every previous difficult result, perhaps making the continuation of the session impossible. Therefore, in order to take advantages of all the possibilities Kenzo is offering now on powerful scientific servers, we are faced with some kind of *state sharing* among different computers (the local computers and the server), a problem known as difficult in the field of distributed object-oriented programming.

In short, even if our initial goal was not related to distributed computing, we found that in order to enable our intermediary layer as an *intelligent assistant* with respect to the classification of calculations as simple (runnable on a standard local computer) or complicated (to be sent to a remote server), we should solve problems of distributed systems. Thus, a larger perspective is necessary, and we are working with the *Broker* architectural pattern, see [3], in order to find a natural organization of our intermediary layer.

8 Conclusions

The current state of our project can be considered solid enough to be a good point of continuation for all our objectives. We have showed how some *intelligent guidance* can be achieved in the field of Computational Algebraic Topology, without using standard Artificial Intelligence techniques. The idea is to build an intermediary layer, giving a mediated access to an already-written symbolic computation system. Putting together both Kenzo itself and the intermediary layer, we have produced a framework which is able to be connected to different clients (desktop GUIs, web applications and so on). In addition, with this framework, several profiles of interaction can be considered. In general, this can imply a restriction of the full capabilities of the kernel system, but the interaction with it is easier and enriched, contributing to the objective of increasing the number of users of the system.

References

- [1] Andrés M., Pascual V., Romero A., Rubio J., *Remote Access to a Symbolic Computation System for Algebraic Topology: A Client-Server Approach*, Lecture Notes in Computer Science 3516 (2005) 635–642.
- [2] Ausbrooks R. et al., *Mathematical Markup Language (MathML) Version 2.0* (second edition), 2003. <http://www.w3.org/TR/2003/REC-MathML2-20031021/>.
- [3] Buschmann, F., Meunier, R., Rohnert H., Sommerland P., Stal M., *Pattern-oriented software architecture. A system of patterns, Volume 1*, Wiley, 1996.
- [4] Buswell S., Caprotti O., Carlisle D.P., Dewar M.C., Gaëtano M., Kohlhase M. *OpenMath* Version 2.0, 2004. <http://www.openmath.org/>.

- [5] Calmet, J., Homann, K., *Towards the Mathematics Software Bus*, Theoretical Computer Science 187 (1997) 221–230.
- [6] Carlsson G., Milgram R. J., *Stable homotopy and iterated loop spaces*. in [14], pp 505-583.
- [7] Cordero C. C., De Miguel A. , Domínguez E., Zapata Ml A., *Modelling Interactive Systems: an architecture guided by communication objects in HCI related papers of Interacción 2004*, Springer (2006) 345–357.
- [8] Domínguez C., Lambán L., Rubio J., *Object oriented institutions to specify symbolic computation systems*, Rairo - Theoretical Informatics and Applications 41 (2007) 191-214.
- [9] Domínguez E., Zapata M.A., *Noesis: Towards a situational method engineering technique*, Information Systems 32,2 (2007) 181-222.
- [10] Dousson X., Rubio J., Sergeraert F., Siret Y., *The Kenzo program*. <http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/>
- [11] Franz Inc. *Allegro Common Lisp*. <http://www.franz.com/>.
- [12] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
- [13] González-López M. J., González-Vega L., Pascual A., Callejo E., Recio T., Rubio J., *TutorMates*. <http://www.tutormates.es/>.
- [14] *Handbook of Algebraic Topology* (Edited by I.M. James). North-Holland (1995).
- [15] *Internet Accessible Mathematical Computation (IAMC)*. <http://icm.mcs.kent.edu/research/iamc.html>.
- [16] Lambán L., Pascual V., Rubio J., *An object-oriented interpretation of the EAT system*, Applicable Algebra in Engineering, Communication and Computing 14 (2003) 187–215.
- [17] Object Management Group. *Common Object Request Broker Architecture (CORBA)*. <http://www.omg.org>.
- [18] Rubio J., Sergeraert F., *Constructive Algebraic Topology*, Bulletin des Sciences Mathématiques 126 (2002) 389-412.
- [19] Schelter W., *Maxima*. <http://maxima.sourceforge.net/index.shtml>.
- [20] Sergeraert F., \mathfrak{X}_k , *objet du 3^e type*. Gazette des Mathématiciens, 2000, vol. 86, pp 29-45.
- [21] Stein W., *SAGE mathematical software system*. <http://sage.scipy.org/sage/>.
- [22] Winer D., *Extensible Markup Language-Remote Procedure Call (XML-RPC)*. <http://www.xmlrpc.com>.

Vns - Name Space Facility

Jim Newton – VCAD Cadence Design Systems

May 7, 2008

1 Introduction

The users of Cadence Design Systems custom IC (integrated circuit) tools use the SKILL¹ programming language extensively. Programmers write applications which customize the look and feel of the graphical system, automate the design process by reducing the amount of repetitive work the design engineer must do, and preform time-consuming, tedious verification checks. Other common types of SKILL programs include automatic layout generation which quickly produce parameterizable layouts which are correct by design. The language has an optional C-style syntax with many engineer-friendly shortcuts which makes it easy for non-programmers to write simple scripts to help in their daily work.

The same language is also a lisp system having the features one would expect such as a REPL, a debugger, garbage collection, lexical and dynamic scoping, macros, and lambda functions. As with most lisp systems, the language can be extended though adding functions to the run-time environment. Unfortunately SKILL is missing many advanced features one would expect from a modern lisp.

Symbols in the SKILL language are implemented in terms of a global name space. Programmers manage to implement ad-hoc packages by incorporating symbol-prefix conventions on all global symbols of their applications. All functions and global variables from one *package* must share the package prefix. Thus, different applications do not interfere, at least not in naming.

Several problems arise from these ad-hoc packages:

- They are sufficient for small applications but become ever more clumsy as applications grow. The very long symbol names detract from readability, and often necessitate refactoring when *package* names change.
- Two such *packages* might conflict when naming **anything** designated by a symbol. The most obvious examples are function names and variable names, but other less obvious examples are menu identifiers, type names, class names, slot names, and form field identifiers.
- They do not accommodate object oriented programming very well. All methods of a generic function must have the same name. This means if two different *packages* wish to declare methods on the same generic function, the methods cannot obey the package prefixing.

¹SKILL is a registered trademark of Cadence Design Systems.

The Common Lisp package system provides a mechanism for grouping names (symbols) into name-spaces. The purpose of the **Vns SKILL** application is to provide the SKILL programmer in a limited fashion with some of the niceties of the Common Lisp package system. It does so as a SKILL application itself—without modifying any of the low level SKILL implementation.

In many cases the code which needs to access such symbols is localized to a few functions. If this is the case those functions can be encapsulated in what is called a name-space.

2 Restrictions

There are some considerations about how the SKILL language works which influence what is possible and not in the perspective implementation of a Common Lisp-like package system for SKILL.

- The programmer does not have access to the SKILL implementation. All extensions must be provided in terms of the existing language implementation and limitations.
- Whereas the Common Lisp **READ** function can be influenced by the dynamic environment (e.g., by the value of ***package***) the SKILL **read** function cannot. We cannot prevent symbols from being created and read time. One work-around might be to implement a **VnsRead** function and a corresponding **VnsLoad** function based on **VnsRead**. This approach was not chosen for several reasons:
 1. The SKILL **read** function is not fully documented or specified, so it would be difficult to imitate all its semantics.
 2. Users expect to simply be able to call **load** to load their files.
 3. Other features of the SKILL language call **load** and **read** and we cannot prevent them from doing so.
- There is no facility in SKILL to destroy a symbol once it has been created. Thus, after **read** returns an s-expression containing symbols, the symbols are there for good.
- The SKILL reader translates certain predetermined infix operators into s-expressions. For example a form such as **(a+b+c)** (with or without extra white-space) is read as the list **(plus a b c)**. The programmer cannot create additional infix operators which are not already built into the language, the programmer cannot change the reader rules of existing infix operators, and the programmer cannot prevent this infix interpolation. The reader evokes an error if such infix operators are used incorrectly.
- The colon **(:)** is such an operator. The expressions **foo:bar** and **(foo:bar)** both read as **(range foo bar)**, and an attempt to read an expression such as **foo::bar** evokes a syntax error.
- It was desired to implement a name-space system which is in-keeping with a SKILL-like philosophy, but at the same time taking advantage of some of the Common Lisp package features.

- It was desired that such an implementation not be intrusive to users who do not choose to use it.

3 Implementation

A *name-space* is an instance of the class `VnsPackage`. There is a predefined subclass `VnsGlobalPackage`, of which there is a single instance representing the SKILL name-space of symbols. Application specific name-spaces may be further defined declaratively with `VnsDefPackage`, or by the programmatic interface `VnsEnsurePackage`. A simple call to `VnsDefPackage` with a given *name-space* name creates an instance of the `VnsPackage` class. Subsequent calls to `VnsDefPackage` with the same name modify the instance's content but package identity is maintained. Such subsequent calls to `VnsPackage` are allowed change any of the meta-information about the class such as which packages are *used*, or which symbols are *interned*, *exported*, or *externed*.

Any code that uses package semantics must be wrapped in a `(VnsWithPackage ...)` form. The `VnsWithPackage` macro rewrites the included code and replaces all occurrences of `pkg?symbol` with the fully qualified name for that symbol. Global symbols may be referenced as `t?foo` or `t??foo`.

4 Concepts

A Vns name space is a container for symbol names. To define a package you must specify a name as an unquoted symbol, and you may specify any of several keyword arguments. The possible options are :

- `?export` – list of symbols to export
- `?use` – list of other packages whose exported symbols to import
- `?shadow` – list of symbols to explicitly NOT import from the included *used* packages.
- `?mapping` – explicit mapping of certain symbols.
- `?packageClass` – Name of some subclass of `VnsPackage`.
- `?intern` – List of symbols to immediately intern into the package upon its creation.

E.g.,

```
(VnsDefPackage foo ?use (bar1 bar2 bar3)
      ?shadow (fun1)
      ?export (fun1 class1 class2 name3))
```

4.1 Name-Space Definition

A name space can be defined with the macro `VnsDefPackage`. A name space must have a name which is global. However, all the symbolic names within the name-space are *local*. In the following example the name-space `SampleOption`

declares that the five names `option`, `optionY`, `drawAnode`, `drawCathode`, and `drawConnector` are local names. Only code inside a `(VnsWithPackage SampleOption ...)` form may reference such a symbol by its simple name. Any reference to the short form of a symbol such as `option` refers to a different symbol.

```
(VnsDefPackage SampleOption
  ?use (Slim)

  ?intern (option
           optionY
           drawAnode
           drawCathode
           drawConnector))
```

4.2 Internal Names

Names internal to a name-space can be reference by other name spaces using a special syntax. The symbol `SampleOption?option` references the `option` symbol in the `SampleOption` name-space. However, if the intent of the name-space is to allow other name-spaces to reference its symbols, the name-space should `export` the symbol.

4.3 Exported Names

The `SampleOption` definition contains the keyword `... ?use (Slim) ...`. This allows code lexically inside any `(VnsWithPackage SampleOption ...)` form to also reference the exported symbols of the `Slim` name-space.

Any symbol which is *exported* from a name space can be referenced in another package by its simple name provided there is an appropriate `... ?use ...` in the name-space declaration.

4.4 External Names

All symbols found inside a `(VnsWithPackage ...)` form which are neither internal nor exported are considered *external*. If an attempt is made later to *intern* or *export* such a symbol, a fatal error occurs. There are functions provided for removing symbols from the *external* symbols list.

Warning, this is actually the opposite of the corresponding behavior of Common Lisp, and admittedly is a less than optimal and counter-intuitive limitation. In Common Lisp, the reader interns newly encountered symbols into the `current` package. From the Vns/SKILL perspective, the symbol has already been *interned* into the global symbol table before the `VnsWithPackage` macro evaluates. The symbol may be intended as the name of a global function which has simply not yet been defined. It would be very difficult if not impossible for a SKILL program to figure out whether such a symbol is intended to reference such a global symbol or a local name.

4.5 Used Packages

A name-space (a primary one) may **use** other name-spaces (secondary ones). This means that the symbols exported from the secondary name-space is visible to code in the primary name-space by their abbreviated names, and symbols internal to the secondary name-space may be referenced by a special syntax—not a colon as in Common Lisp but `?` such as `F00?name`. Any symbol in any existing package may be referenced within a `VnsWithPackage` form with the syntax `??` such as `F00??name`.

The global name-space `t` is analogous to the Common Lisp `CL` package. References to global symbols may be made such as to the global names `car` and `defclass` may be made as `t?car` and `t?defclass` because every name-space implicitly *uses* the `t` name-space.

4.6 Shadowed Symbols

When a primary package *uses* a secondary package which exports a particular symbol, it is possible to specify that that particular symbol not be imported into the package. Such symbols are called *shadowed*. They may still be referenced as in the following example.

```
(VnsDefPackage F00
  ?export (name1 name2))
(VnsDefPackage BAR
  ?shadow (name1)
  ?intern (name1))

(VnsWithPackage BAR
  (list 'name1      ;; represents name1 in BAR
        'F00?name1 ;; represents name1 in F00
        'name2     ;; represents name2 in F00
  ))
```

4.7 Name Mapping

Since the SKILL language implementation does not know anything about these `Vns` name-spaces. Thus the code included in `(VnsWithPackage ...)` translates to valid SKILL code using normal SKILL symbols. The method `VnsQualifySymbol` maps a symbol semantically in a name-space into the global SKILL name space.

By default the `VnsQualifySymbol` function creates global symbols which are not very friendly to the human eye. These symbols are not intended for human consumption but rather to be referenced by other packages by their friendlier names. Sometimes it is desired for a symbol to have a human-eye-friendly spelling so it can be used by a programmer who is not using the **Vns** package system. In such cases a symbol mapping can be specified. For example, one can specify that a symbol be represented as `foo` inside a `VnsWithPackage` form but as `MyAppFoo` to other applications.

Such a mapping can be specified explicitly with the `?mapping` keyword to `VnsDefPackage` or by the `?export` keyword. The argument to `?export` is a list of symbols and symbol pairs. Symbols represent names to be exported according

to the mapping specified by `VnsQualifySymbol`, and symbol pairs represent the internal and external name.

Obviously, conflicts can occur if two different packages attempt to export two different internal symbols as the same global symbol. It is intended that the programmer take the same care in naming these global symbols as would be necessary without the Vns system in use at all.

5 Examples

The following form defines a name-space. The primary purpose of a name-space is to prevent symbols from polluting the SKILL global name space. However, a nice side effect is that SKILL code written inside that name-space need not use long and silly prefixes which detract from readability and portability.

```
(VnsDefPackage SampleOption
  ?use (Slim)

  ?intern (option
           optionY
           drawAnode
           drawCathode
           drawConnector))
```

The `VnsDefPackage` specifies several things:

- The name of the name-space: `SampleOption` in this case.
- Zero or other name spaces to *use*: `Slim` in this case.
- Which new symbols to protect: `option`, `optionY`, `drawAnode`, `drawCathode`, and `drawConnector`.

Once a package has been defined, it may be used none, once, or many times with the `VnsWithPackage` macro. Within the `VnsWithPackage` form references to symbols may be made using the `?` character: `pkg?name`. A reference to the current package such as `(VnsWithPackage FOO ... FOO?name ...)` automatically interns name into the FOO package, unless a conflict is detected.

E.g., if the name has been used in the `VnsWithPackage` form prior to its usage in `FOO?name`. However, a more explicit and more readable way to perform this interning is with the `?intern` keyword parameter of `VnsWithPackage`.

E.g., `(VnsWithPackage foo ?intern (name) ... name ...)` Inside a `VnsWithPackage` form for one package, you may reference symbols in another package with the `?` separator, whether or not they are exported from that package, and you may reference names exported from the any included `(?use ...)` package simply by name.

E.g, `(VnsDefPackage foo ?export (fun1))`

```
(VnsWithPackage foo
  (defun fun1 () ;; define fun1 in package foo
    100)
```

```

    (defun fun2 (x) ;; define fun2 in package foo
      x+200))

(VnsDefPackage bar ;; define package bar
  ?use (foo) ;; include all the exported symbols from foo
  ?export (fun3) ;; also export fun3 from bar

(VnsWithPackage bar
  (defun fun3 ()
    (list (fun1) ;; fun1 from package foo can be called by name
           ;; because foo is imported with ?use (foo) and
           ;; fun1 is exported from foo.
          (foo?fun2 0) ;; fun2 must be referenced with foo:fun2
           ;; because fun2 is not exported from foo
    )))

```

5.1 Syntax Examples

Example 1: Interns name into package FOO

```

(VnsWithPackage FOO
  ...
  FOO?name
  ...
)

```

Example 2: References an exported symbol from another package. Error if name is not exported from FOO.

```

(VnsWithPackage BAR
  ...
  FOO?name
  ...
)

```

Example 3: References an interned symbol in another package. Error if no such symbol is interned in that package.

```

(VnsWithPackage BAR
  ...
  FOO??name
)

```

Example 4: Export a symbol with a given global name.

```

(VnsDefPackage FOO
  ?export ((bar VcadGvBar)))

```

This allows the name of the global variable `VcadGvVar` to be used inside package `FOO` simply as `bar`. Other packages may reference it either by its advertised name `VcadGvBar` or by `FOO?bar`.

6 Open Issues

1. Subclasses of `VnsPackage` are free to override the `VnsQualifySymbol` method. It is unclear at the current time as to whether this is useful. If such functionality is desired, it must be made clear what the procedure is. For example `VnsLookUpSymbol` must also be implemented to as an inverse function. Also other methods such as `VnsDecomposeName` and `VnsExtractName` must be documented.
2. Name-space redefinition semantics need to be flushed out. It is not clear to me from reading the Common Lisp specification what the exact semantics of package redefinition are. What happens to symbols which are already interned or external to a package which it the package is redefined, or if one of the dependent packages is redefined.
3. Interactive editors such as emacs, cannot lexically look at code and correctly determine what the symbols real names are. This causes difficulty when trying to trace functions by their printed names.
4. There are several features missing from the Common Lisp package system which would be interesting to experiment with. One such feature would be the ability to *use* two different *versions* of the same package simultaneously. Such a capability would allow to different packages to coincide even though they both require different versions of some third base package.
5. Another area of potentially interesting investigation is the definition of the package Meta-object protocol which is missing from Common Lisp. I.e., the behavior of the Vns system is based on generic functions which can be overwritten for different subclasses of `VnsPackage`.

7 Conclusion

It seems from a bit of experience using the `VnsPackage` system experimentally that SKILL code written in such a system is cleaner and easier to read. However, the implications are unclear. It is unclear whether such an add-on name-space system has hidden gotchas that will burden the programmer.

Additional research and experimentation is needed.

Prime-Lisp 2.0: an ISLisp Implementation in .NET with Multithreading Extensions

Mikhail Semenov

(TMA Data Management, Kington-upon-Thames, England,
mikhail.semenov@tma.co.uk)

Abstract: Prime-Lisp 2.0 is practically a full version of the ISLisp standard, which is being implemented in C#. It going to include several extensions to the standard: multithreading, graphics, algebraic routines. The intention is to be able to use the rich .NET environment. A relatively easy and low-costly multithreading in .NET makes it possible to revive some of the features of Lisp parallelism that were designed in QLisp and other Lisp dialects. The results of benchmarks are given. Other possible extensions and trends are discussed as well.

Keywords: ISLisp, C#, .NET, multithreading, parallelism, graphics, symbolic computation

Categories: D.1, D.3.3, I.1.3, I.2.8, I.3

1 Introduction

As .NET is getting popularity with programmers, and languages like Python [IronPython, 07] have been implemented in .NET, it seems natural to develop a Lisp implementation. There have been some successful attempts in doing that (DotLisp [Hickey, 03] and Lisp Sharp [Blackwell, 06]). Out of these two, Lisp Sharp seems to have a good approach to implementation and easy access to native .NET classes. But both lack a lot of standard features and deviate from the existing standard dialects. For example, Lisp Sharp uses the “=” for assignment and “==” for comparison (like C, C++ and C#).

Eventually, the decision was made to provide a new implementation in C#, which would be based on a standard. So the ISLisp [ISLisp, 07] standard was chosen. The C# environment will make it possible to access .NET features via a foreign-language interface. This is how Prime-Lisp 2.0 was born.

There is an intention to provide the following extensions: graphics (mainly 2D graphics to display graphs and images), and numeric extensions.

Traditionally parallelism naturally occurs in Lisp. In pure Lisp (with no side-effects), the arguments of functions can all be computed in parallel without any changes in the results. These features are discussed widely in literature, for example in [Goldman, 89], [Yuen, 93]. There are a few multithreading extensions of Common Lisp [CommonLisp, 94], explained in [Cracauer, 07], [Clementson, 05]. It is quite convenient and easy to create and use light-weight multithreading in Lisp. In Prime-Lisp 1.0 [Semenov, 94], a Lisp implementation was discussed that used an extension to the multi-value function mechanism, which provides parallelism. The idea is rather simple: at certain points in code the several values that are generated by a multi-value function create several threads of computation, in each of them a particular value of a function being used. Such approach can create a “swarm” of threads. There can be a function call, which embraces the multithreading code, that will collect all the values produced by various threads into a list, which can be used for future examination. Obviously a mechanism for cutting some threads and stopping execution of those that

have executed too long can be provided. At the time of Prime-Lisp 1.0, light-weight multithreading was not available and the multi-value function mechanism was implemented inefficiently. A new version, Prime-Lisp 2.0, was conceived in 2008 and it will contain most of the ISLisp standard [ISLisp, 07] with minor exceptions.

2 Already Implemented ISLisp Features

At the time of this publication the following features of ISLisp have been implemented.

2.1 Standard Data Types

The data types (classes) that have been implemented so far are listed in Table 1. It is also mentioned whether a standard C# (.NET) data type is used or a new one is defined.

ISLisp Type	Internal Implementation	Standard/Defined
<symbol>	LispSymbol	Defined
<character>	char	Standard
<integer>	long	Standard
<float>	double	Standard
<string>	StringVector	Defined
<list>	LispList	Defined
<general-vector>	object[]	Standard
zero-dimensional <general-array*>	ZeroDimArray	Defined
multidimensional <general-array*>	object[,...]	Standard
<function>	FunctionValue	Defined
user-defined class	UserDefinedClass	Defined
used-defined object	UserDefinedObject	Defined
<error>	LispError	Defined, but based on the .NET Exception class.

Table 1: ISLisp Types Implemented in Prime-Lisp

Other types (such as streams), which are not mentioned in this table, are yet to be implemented and probably will use standard .NET stream classes. In Prime-Lisp, there is an additional class, called <window>, which is discussed in section 4.

The strings are mutable, as well as in other Lisp dialects (ISLisp, Common Lisp).

2.2 Syntax

The syntax is the same as in ISLisp. Particular attention has been paid to the syntax of symbols, strings and characters. The backquote is supported. In addition, it is possible to use the feature that will convert processed backquote syntax (usually expressions using **append**) back into backquote representation. In this syntax, (**append** x y) will look like this: ``(,@x ,@y)` . All the backquote examples from [ISLisp, 07] look the same in Prime-Lisp 2.0. In section 3 we will be discussing optional extensions to the traditional Lisp syntax.

2.3 Standard Functions

The function **syntax**, **lambda** expressions, **labels**, **flet**, **defun**, **defconstant**, **defdynamic**, **defglobal**, **defmacro**, **apply** and **funcall** have been implemented exactly as in the standard.

The **setf** syntax has been implemented for all the standard functions (**car**, **cdr**, **dynamic**, **aref**, **garef**, **elt**, **property**). There is a mechanism in place to allow it for user-defined functions as well.

Block functions, loops and various jump mechanisms (**let**, **let***, **dynamic-let**, **progn**, **block**, **return-from**, **tagbody**, **go**, **catch**, **throw**, **for** and **while**) have been implemented.

Basic list-processing functions exist: **car**, **cdr**, **cons**, **list**, **reverse**, **member**, **assoc**, **nreverse** and **append**. Functions for sequences and arrays have been implemented: **elt**, **aref**, **garef**, **create-list**, **create-array**, **create-vector**, **create-string**, **vector**, **length**, **subseq**. All the string functions are working: **char-index**, **string-index**, **string-append**. The **convert** function, which works with various types, has been implemented.

Comparison operators and logical and conditional functions have been implemented: **eq**, **eql**, **equal**, **=**, **/=**, **<**, **>**, **<=**, **>=**, **string=**, **string/=**, **string<**, **string>**, **string<=**, **string>=**, **char=**, **char/=**, **char<**, **char>**, **char<=**, **char>=**, **not**, **and**, **or**, **cond**, **if**, **case**, **case-using**.

All the arithmetic functions are working: **+**, **-**, *****, **/**, **quotient**, **reciprocal**, **sin**, **cos**, **tan**, **atan**, **atan2**, **sinh**, **cosh**, **tanh**, **atanh**, **sqrt**, **isqrt**, **abs**, **exp**, **log**, **expt**, **min**, **max**, **div**, **mod**, **gcd**, **lcm**, **float**, **integer**.

All the map functions exist: **mapcar**, **mapc**, **maplist**, **mapl**, **mapcan**, **mapcon**, **map-into**.

As it was mentioned, streams are yet to be implemented. But additional functions such as **print** and **load** are present; they exist in many Lisp dialects, but are not part of the ISLisp standard. The function **format** for the standard output stream has been implemented.

The functions **defclass** and **create**, which allow to create classes and objects, have been implemented. Particular attention has been paid to the keywords in the slots. The accessors, readers and writers are working.

The **defgeneric** and **defmethod** are not implemented yet.

3 Extensions

Here we will be discussing extensions that are already present in Prime-Lisp 2.0.

3.1 Syntax extensions

As it was mentioned before, Prime-Lisp 2.0 uses the standard ISLisp syntax. But it is possible to switch on the optional syntax extensions, which use the curly and square brackets. These extensions can be switched on or off by using the following function:

```
(syntax-extensions arg)
```

If the value of *arg* is nil the function will switch the extensions off; otherwise it will switch them on. When the extensions are switched on their use is still optional, but the meaning of square and curly brackets changes: they cannot be used as symbols, unless escape characters are provided.

3.1.1 The Curly Bracket “Symmetric Syntax”

This syntax can be used for any function call and is particularly important for defining forms (like `defun`), block functions and loops. Any list $(f e_1 e_2 \dots e_n)$, where f is a symbol, can be written as $\{f e_1 e_2 \dots e_n f\}$, which means that the function call can be enclosed in curly brackets, but in this case the function name should be repeated before the close curly bracket. It does not affect the internal representation of the function call. This syntax is optional (even when it is switched on), it can be chosen wherever needed and makes it easier to read and write Lisp programs.

For example, if the following expression:

```
(defun bar (x y)
  (let ((foo #'car))
    (let ((result
          (block bl
            (setq foo
              (lambda () (return-from
                        bl 'first-exit)))
            (if x (return-from bl
                              'second-exit) 'third-exit))))
      (if y (funcall foo) nil)
      result)))
```

can be rewritten like this:

```
{defun bar (x y)
  {let((foo #'car))
    {let({result
          {block bl
            {setq foo
              (lambda () (return-from
                          bl 'first-exit))
              setq}
            (if x (return-from bl
                          'second-exit) 'third-exit)
              block}
            result})
      (if y (funcall foo) nil)
      result
    }
  }
}
defun}
```

3.1.2 The Bracket Syntax for Vectors

This is more traditional syntax: the vectors are more visible when enclosed in square brackets. Steele in [Steele, 90] (page 33), suggested that the square bracket notation “would be shorted, perhaps more readable...”. The reason it was not adopted for Common Lisp was to allow the use of square brackets for something else. Perhaps, it was a bit unreasonable to do that as the expense of clarity. In Prime-Lisp 2.0, this notation is optional. For example, the following vector of vectors:

```
##(##(1 2 3) ##(4 5 6)) ##(##(67 88 99) ##(101 234 788)))
```

can be written like this:

```
[[[1 2 3] [4 5 6]] [[67 88 99] [101 234 788]]]
```

3.2 Multithreading

Here we are going to discuss creation of threads which execute in parallel. There are two main approaches in Prime-Lisp 2.0: independent-thread creation and thread collection. The first deals with creating a separate thread that executes independently, the second, with evaluation of several expressions in parallel.

3.2.1 Independent-Thread Creation

The following function will create a thread:

```
(create-thread e) => thread object
```

This function will create a thread that will evaluate the expression **e**. The result of the function (which is produced immediately) is the thread object that has been created. By referencing the thread object it is possible to get the status of the thread and the result. The following function gets the status of a thread:

```
(thread-status thread-object)
```

The values of this function (status values) can be as follows:

- **running**, this symbol means that the thread is running;
- **(finished result)**, which means that the expression has been successfully evaluated and the result has been produced (the second element of the list);
- **(error error-value)**, the execution stopped due to an error;
- **aborted**, this symbol means that the thread has been aborted;
- **waiting**, the thread is waiting.

The following function aborts the execution of a thread:

```
(abort-thread thread-object)
```

It returns **t**, in case of a success (the thread was running at the time), and **nil**, otherwise.

3.2.2 Thread Collection

The following special form is used to create several threads, whose results can be combined into one vector of values:

```
(thread-collection (?n ?tmin ?tmax) p1 p2 ... pn)
```

Here a question mark (?) means that the corresponding expression is optional. If all the optional expressions are absent (the first parameter is **nil**), the parameters **p1**, **p2**, ..., **pn** are executed as separate threads and their statuses are combined into a vector which is returned as a result of the function call. If all the optional parameters are present they should be evaluated first. Their results influence the execution as follows (listed in the descending order of priorities):

- **tmax** limits the time of the execution in seconds (it can be a floating-point value and can define the limit in fractions of a second); when the time of the execution reaches **tmax** the threads that have not finished will be aborted and the statuses of all the threads will be combined into a vector;

- **n** determines the minimum number of the results that should be obtained (minimum number of successful threads);
- **tmin** determines the minimum amount of time the threads should execute (unless all of them have produced values).

In short, the threads will run until at least **n** of them have produced results, but not less than **tmin** and no more than **tmax** seconds; if all the threads have finished early the **tmin** is ignored.

Here are some examples. The execution of function **(run1 m)** takes longer the greater the parameter **m**. The result is the time of execution in seconds (the time might vary a bit, depending on the garbage collection timing).

```
(thread-collection () (run1 20) (run1 15) (run1 10))
=> #((finished 16.750) (finished 0.578) (finished 0.031))

(thread-collection (2 3 10) (run1 20) (run1 15) (run1 10))
=> #(aborted (finished 0.547) (finished 0.016))

(thread-collection (3 0 10) (run1 20) (run1 15) (run1 10))
=> #(aborted (finished 0.547) (finished 0.016))

(thread-collection (3 0.7) (run1 20) (run1 15) (run1 10))
=> #((finished 16.750) (finished 0.578) (finished 0.031))

(thread-collection (2) (run1 20) (run1 15) (run1 10))
=> #(aborted (finished 0.547) (finished 0.016))
```

The **thread-collection** function can be used in applications, where various complex algorithms are used to find a solution to a problem. Some of the algorithms can produce infinite loops, but others will be able to find a solution. It is like finding a way in a maze: some paths may produce loops, while other will lead to a way out.

3.2.3 Thread Synchronisation

The following special form establishes a lock when a function's parameters and body are evaluated:

```
(exclusive-funcall f p1 p2 ... pn)
```

The first parameter is a function value (**#'function-name** or a **lambda-expression**). This **exclusive-funcall** function makes sure the parameters and the body of the function **f** are executed in one thread at a time. If several threads try to use **exclusive-funcall** with the same first parameter they will be suspended until the function **f** is free.

The following function suspends the execution of the current thread until the given threads have finished:

```
(thread-wait n t1 t2 ... tn)
```

The first parameter *n* (a float) defines the maximum amount of seconds to wait. The values of *t1 t2 ... tn* should be thread objects. The current thread waits at most *n* seconds for the threads to finish. After that it aborts all of them and resumes its execution.

The following function suspends the execution of the current thread for *s* seconds:

```
(thread-sleep s)
```

The value *s* can be a floating-point number. Let us look at the following example:

```
(defglobal counter ())
(defun count(x y s)
  (cond ((>= (length counter) 20) ())
        (t
         (setq counter (cons x counter))
         (thread-sleep s)
         (setq counter (cons y counter)))))

(defglobal a nil)
(defglobal b nil)

(defun run1()
  (setq counter ())
  (setq a (create-thread (while (count 'A 'A1 2))))
  (setq b (create-thread (while (count 'B 'B1 3)))))

(defun run2()
  (setq counter ())
  (setq a(create-thread
          (while(exclusive-funcall #'count 'A 'A1 2))))
  (setq b (create-thread
          (while(exclusive-funcall #'count 'B 'B1 3)))))
```

The following results may be produced:

```
(progn (run1) (thread-wait 1000 a b) counter)
=>(b1 a1 b a b1 a1 a a1 b b1 a a1 a a1 a a1 b b1 a a1 b a)

(progn (run2) (thread-wait 1000 a b) counter)
=> (b1 b a1 a b1 b a1 a b1 b a1 a b1 b a1 a b1 b a1 a)
```

In the result of the first **progn**, the exact amount of elements in the list may vary because the threads' execution overlap. But the important issue is that the use of **exclusive-funcall** in the second example ensures that **a1** is always pushed into the list after **a** and **b1** is pushed after **b**.

4 Graphics

The following features are implemented to support graphics. In order to create a graphic window (which is called a form in .NET) this function is used:

```
(create-window title x0 y0 width height colour scalability)
```

The function creates a window (which belongs to the class <window>); **title** is a text string displayed in the caption; **x0** and **y0** are the co-ordinates of the top left corner of the window; **width** and **height** are its initial width and height; the parameter **colour** defines the background colour of the window; **scalability** determines whether (if not **nil**) the graphics in the window is scalable when the window is resized or (if **nil**) the window is of fixed size. The value of the function is the created window. When a window is created it is immediately displayed.

In order to provide various graphics functionality for created windows special methods are used. All such methods have a window as their first parameter, which means that potentially it is possible to define methods with the same name for a different class.

The following methods are provided:

- **(close window)**, which closes the window;
- **(line window colour thickness x1 y1 x2 y2 ... xn yn)**, which draws a zigzag line;
- **(text window font-name font-size colour orientation x y)**, which displays a text string;
- **(fill-rectangle window colour x0 y0 width height)**, which draws a solid rectangle;
- **(rectangle window colour thickness x0 y0 width height)**, which draws the outline of a rectangle;
- **(fill-ellipse window colour x0 y0 width height)**, which draws a solid ellipse;
- **(ellipse window colour thickness x0 y0 width height)**, which draws the outline of an ellipse;
- **(fill-pie window colour x0 y0 width height angle range)**, which draws a solid sector of a circle starting with the given **angle** and finishing with the angle **angle+range-1**;
- **(arc window colour thickness x0 y0 width height angle range)**, which draws an arc of a circle starting with the given **angle** and finishing with the angle **angle+range-1**;
- **(mouse-position window font-name font-size text-colour x11 x12 y11 y12 x21 x22 y21 y22)**, which allows to determine the mouse position when the left mouse button is pressed; where **x11**, **x12**, **y11** and **y12** define the parameters of the rectangle in the original window co-ordinates; **x21** **x22** **y21** and **y22** define the corresponding rectangle, whose points will be used to display the mouse position (these values can be floating-point).

Here is an example, which creates a window shown in Figure 1:

```
(defglobal w (create-window "graph test" 0 0 400 200 #x00 t))
(line w #xFFFFF00 2 10 5 50 100 100 50)
(text w " Graphics" "Times New Roman| italic bold" 6 #xFF00 0 100 100)
(text w " Graphics" "Times New Roman| italic bold" 6 #xAF0000 -90 100 100)
(text w " Graphics" "Times New Roman| italic bold" 6 #xFF1F00 45 100 100)
(text w " Graphics" "Times New Roman| italic bold" 6 #x001FFF -45 100 100)
(fill-rectangle w #xFF7F00 10 100 40 30)
(fill-ellipse w #x7F00FF 50 100 40 30)
(rectangle w #xFF0FFF 1 10 100 40 30)
(ellipse w #xFFFFFFFF 2 50 100 40 30)
(fill-pie w #FFFFFF00 50 100 40 30 30 60)
(arc w #xFF 10 50 100 40 30 -90 45)
(for ((i 0 (+ i 1))) (= i 200) nil)
  (let* ((step (/ *pi* 50)) (x (* step i)) (x2 (+ x step)))
    (line w #xFF007F 1 (+ 150 i) (+ 100 (* 7 (* (sin x) x)))
          (+ 150 i 1) (+ 100 (* 7 (* (sin x2) x2))))))
```

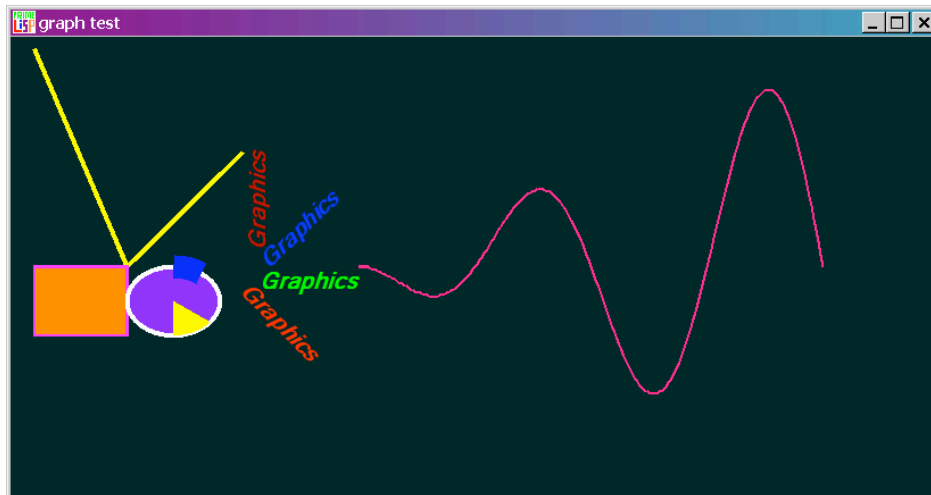


Figure 1: A graphics example

5 Benchmarks

The original OpenLisp [OpenLisp, 08] port of Gabriel's benchmarks were used. Two tests Hanoi_Towers1 and Hanoi_Towers2 were added. The results are shown in Table 2.

Benchmark	Prime-Lisp	OpenLisp
Fib	0.062	0.003
Tak	0.219	0.011
Stak	0.265	0.017
Takl	0.219	0.017
Takr	0.187	0.013
Boyer	2.797	0.367
Browse	2.235	0.252
Destru	0.391	0.04
Travini	3.016	0.188
Travrun	13.468	1.078
Deriv	0.266	0.044
Dderiv	0.312	0.039
Divit	0.266	0.035
Divrec	0.375	0.029
FFT	1.703	0.152
Puzzle	3.469	0.194
Triang	41.61	3.24
Hanoi_Towers1 (20)	14.844	3.497
Hanoi_Towers2 (20)	13.047	0.933
TOTAL	98.751	10.149

Table 2: Benchmarks. The timings are given in seconds.

The Towers of Hanoi problem is described in [Tower, 08]; it has the following algorithm in Lisp:

```
(defun hanoi(n a b c)
  (cond((= n 0) ())
        (t
         (append
          (hanoi (- n 1) a c b)
          (list(list 'move n a b))
          (hanoi (- n 1) c b a))))))
```

Hanoi_Towers1 uses the exact code shown above, and **Hanoi_Towers2** uses destructive **nconc** instead of the **append**. Both tests use 20 disks ($n = 20$).

As the tests show, the speed of Prime-Lisp is one tenth of the speed of OpenLisp. There are some tests that have been excluded: they deal with the **go** function. The implementation of **go** in Prime-Lisp relies on the exception mechanism of .NET, which is not fast. Other excluded tests involve streams and formatted input-output, which are not yet fully implemented in Prime-Lisp.

At the same time, when Prime-Lisp was tested against free editions of a few Lisp interpreters, it runs faster.

6 Future Plans

Here we are going to discuss future development: issues that will be implemented soon and those that are under consideration.

6.1 The ISLisp Features That Are To Be Implemented

The plan is to finish the implementation of the ISLisp standard:

- generic functions and methods;
- streams and formatted input-output;
- error messages and error processing.

There is a temptation to implement a cut-down version for methods that will permit methods to work only with user-defined classes and will allow to specify only one (the first) typed parameter for a method, similar to methods for the <window> class. That will provide a faster implementation, which will be more in line with other OOP languages, but may be considered to primitive for a Lisp implementation.

6.2 Numeric Extensions

In the nearest future there is a plan to provide some basic features for matrix and vector operations. It is possible that the arithmetic operators (+,-,*) will be extended to cover vectors and multidimensional arrays as well.

As for BigNums, we have developed code in C++ and C# that allows multiple precision arithmetic (including standard and some special functions). There is an intention to include it into Prime-Lisp. But it will be more useful for floating-point arithmetic, and will probably be not that fast for integer and rational arithmetic.

6.3 The Foreign-Language Interface

There will definitely be an interface, which will allow to include code developed in C# (or any other .NET language), mainly through the use of .NET DLLs. That will, in turn, provide access to code written in C++ as well.

References

- [Blackwell, 06] Blackwell, R.: Lisp Sharp. L Sharp .NET - A powerful lisp-based scripting language for .NET, <http://www.lsharp.org/>
- [Clementson, 05] Clementson, B.: Parallel Computing in Lisp – Part 3, <http://bc.tech.coop/blog/050119.html>
- [CommonLisp, 94] ANSI INCITS 226-1994, Programming Language Common Lisp.
- [Cracauer, 07] Cracauer, M.: Thread Interfaces in Common Lisp, <http://www.cons.org/cracauer/lisp-threads.html>
- [Goldman, 89] Goldman, R., Gabriel, R.P.: Qlisp: Parallel Processing in Lisp, IEEE Software, July/August, 1989, V. 6, N. 4, pp. 51-59.
- [Hickey, 03] Hickey, R.: DotLisp – A Lisp Dialect for .NET, <http://dotlisp.sourceforge.net/dotlisp.htm>
- [IronPython, 07] .NET IronPython, <http://www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython>
- [ISLisp, 07] ISO/IEC 13816:2007 - Programming Language ISLISP..
- [OpenLisp, 08] OpenLisp by Eligis, <http://christian.jullien.free.fr>
- [Semenov, 1994] Semenov, M.: The Integrated Windows Environment of PRIME-LISP, Proceedings of the Fourth International Lisp Users and Vendors Conference, August 15-19, 1994, pp. 1-9.
- [Steele, 90] Steele, G.L.: COMMON LISP. The Language. 2nd Edition. Digital Press, 1990.
- [Tower, 08] Tower of Hanoi, http://en.wikipedia.org/wiki/Tower_of_Hanoi
- [Yuen, 93] Yen, C.K, et al.: Parallel Lisp Systems. CHAPMAN & HALL, 1993.

Abolishing object-oriented xenophobia: designing highly reusable libraries

Pierre Thierry
(Independent developer
Thierry Technologies
4, rue de Fegersheim
Strasbourg, France
pierre@thierry-technologies.com)

Simon E.B. Thierry
(Doctoral researcher
LSIIT, UMR CNRS-ULP 7005
Université de Strasbourg Bd Sébastien Brant, Illkirch, France
thierry@lsiit.u-strasbg.fr)

Abstract: Programming reusable libraries has in the past few decades become a solution to an increasing crisis in software engineering. Indeed, the use of programming libraries allows a developer to focus on his task rather than re-implementing classical algorithms and tools, but their strictness often prevents using them in a role they were not meant for. The goal of reusable libraries is to be as adaptable as possible, so as never to block the developer who uses them for an original task the library designers did not foresee.

In this article, we show that the easiest way for a library to set minimal constraints on its use cases is to use higher-order whenever possible and set no other constraints on the function parameters than their type. Thus we advocate abolishing discrimination of parameter objects by their ancestors to promote discrimination by their abilities, effectively abolishing object-oriented xenophobia.

We show how important it is that library designers bear this in mind by explaining the effects it has on the use range of such libraries. We also show that a lack of reusability in libraries may affect the usability of the language itself.

Key Words: programming libraries, code reuse, higher order

Category: D.1.5, D.2.2, D.2.13, D.3.3

1 Introduction

The terms of *software crisis* are attributed to F.L. Bauer, who used them during the first NATO Software Engineering Conference in 1968 [1]. They describe the problem of writing correct, maintainable and reusable code without running over-time or over-budget.

Various aspects of this crisis have been studied over the years by great figures of the computer science. The writing of understandable code has quickly found solutions so that nowadays developers have all the tools they need to create

programs that someone else can read and maintain. The main stumbling block yet remains to convince programmers of the importance of using such tools.

The automatic or semi-automatic verification of code correctness, on the other hand, is still an active field of research. Various approaches have been proposed, such as Hoare's logic [18], model checking [5], formal refinement methods such as B [2] or interactive inductive proof assistants such as Coq [4].

In this article, we focus on the third aspect of the software crisis: the ways to write reusable code. The issue, there, is to be able to efficiently and quickly include existing bits of programs in a new source code, so as not to reinvent, or in this particular case reimplement, the wheel. It is, of course, essential that the reused parts as well as the resulting whole code preserve their correctness and maintainability properties in the inclusion process.

A popular approach is the use of IPC (Inter-Process Communication) such as sockets, pipes or message queues. IPCs allow the creation of new processes in order to execute another program without losing the ability to have informations transitting between the new process and its parent. The programmer may then reuse existing code without altering it, thus conserving its correctness and without taking time to understand its inner mechanisms. A similar idea lead to the creation of IDL (Interface Description Languages) such as CORBA [14].

The main problem of such approaches is the impossibility, for the developer reusing an existing program, to change its behaviour. If the developer wants to use a program for a different purpose than what it was first meant for, he may encounter difficulties or even impossibilities because the program has too many implicit conditions on the task it is used for.

Another approach is the use of programming libraries: sets of tools and functions designed explicitly to be included by different programs so that the developer does not need to reimplement well-known algorithms from scratch. Here again, the developer may encounter a few difficulties. The first one, which will not be adressed further in this paper, is the language barrier, preventing a Java programmer to easily and efficiently use a Haskell library, although some couples of languages are made usable together by Foreign Functions Interfaces.

The second one is exactly the same as for IPCs: library developers generally intend their code for a specific purpose and, even when they want to make it adaptable, unintentionnally set use conditions by not imagining possible uses of the library. The problem here is actually worst than for IPCs, since external programs can generally be called with enough parameters to ensure a good degree of adaptability. The issue we adress in this paper is the reusability of programming libraries. We believe that, in order to be highly reusable, a library needs to be configurable.

We identified four levels of configurability:

1. the first level exhibits no degree of configurability; examples include a lazy

list (*i.e.* a possibly infinite list where the next value is computed only upon request) of the Fibonacci numbers or a seeded pseudo-random number generator (PRNG).

2. the second level is configurability by a limited set of options; examples include a range generator that can return its result value as a singly-linked list, a doubly-linked list or an array.
3. the third level is configurability by an unlimited set of values; examples include a polynomial function of a fixed order whose coefficients are supplied by the user.
4. the fourth and last level is configurability by policy; examples include a heap container whose ordering is defined by the user. In this context, policy means arbitrary behaviour (and thus, code).

In this article, we explain guidelines for libraries developers to ensure their code is highly reusable. We show that using higher-order programming, considering a library as a couple \langle signature, model \rangle and only enforcing the library parameters to respect signature constraints are three sufficient conditions to reach the fourth level of configurability and thus achieve a high degree of adaptability.

Throughout this paper, the term *object* is used in a wide meaning, and may refer to functions, including closures. Although many programming languages consider objects and function as being fundamentally different, it is merely an implementation artifact and objects and closures have been shown to be strictly equivalent [8].

The rest of this paper is organized as follows: section 2 describes other approaches towards reusable code; section 3 shows the advantages of higher-order programming in terms of configurability; section 4 briefly recalls elements of the algebraic specifications theory that will help us express adaptability conditions and shows the benefits of higher-order for reusable libraries; section 5 details the application of our guidelines to a case study; section 6 shows a few drawbacks of our approach and perspectives to extend our works.

2 Related works

The problem of code reuse has been thoroughly studied, yet not comprehensively. Among the various approaches that have been proposed, some deserve particular attention.

The ADA community, in particular, had set as one of its goal to solve this critical issue related to the “software crisis”.

The difficulty to identify and find existing reusable components, in particular, was studied extensively, and code repositories were designed to alleviate the difficulty [6,19] as well as tools to assist the programmer in the task of producing, distributing and using reusable components [13,25].

The conditions enabling reuse were also investigated. In particular, the code itself may not be the deciding factor of its own reuse, as shown by [26] or [22]. Tools supporting automated or semi-automated classification and search of reusable components were produced to make the process of code reuse practically efficient [7,17]. Code reuse has been shown to be a valuable part of the Quality Assurance of programming [29].

Another important problem is the possibility to adapt an existing code to a new task, which is similar yet not equivalent, to the original problem the program was meant to solve. A recent development in this field is aspect-oriented programming [21,27]. The idea behind aspects is to add a new program module, which will be plugged at possibly many different locations of the original code, modifying its behaviour. The aspect comes with a conditional trigger, which indicates whether the code in the aspect needs to be executed after a variable modification. Main issues to be resolved in the field of aspect programming are performance, a significant loss being produced by the many trigger tests, and readability and certifications of the resulting code.

Recently, the Haskell community also specifically studied the old issue of composability [3,15], that is the property that libraries do not to interfere with each other when used together. Monads have proven to be a critical tool to achieve this goal [11,16,23].

The problem of library reusability has also been studied in the past few years. Fowler *et al.* [9] indicated general conditions for a library to be reusable. Our paper addresses points that Fowler *et al.* referenced as modularity, minimality and evolvability. We actually show that the last two concepts are tightly linked.

3 Higher-order programming

The concept of higher-order is that of an entity operating on its own kind. Classical examples in the domain of computer science are higher-order logic – a logic able to express operations on logic itself – and higher-order functions – functions whose parameters and return values may be functions. The latter play a key role in functional programming, where they are a source of extensibility and reusability.

A typical example of a higher-order function in programming, whose higher-order design makes it both efficient and reusable, is Lisp's *sort* function. It takes three arguments: the sequence to be sorted, the *key* function and the *predicate* function. This design achieves a high degree of reusability, even for

other functions than *sort* itself. The *key* function not only is a gain of efficiency, as the value on which the ordering is calculated is computed outside the ordering, and thus can be computed only once for each element, but also makes it possible to reuse existing ordering functions, if the data to be sorted can be coerced to the type of the ordering function's arguments.

Without designating it as such, the object-oriented programming community already uses higher-order programming. *Stricto sensu*, many design patterns [12] could already be described as higher-order objects, as they operate on objects and return objects (in particular, patterns like the *factory* – an object used to create objects – or the *adapter* – an object that calls its encapsulated object for most of its methods). But there is a design pattern in particular that extends object-oriented programming the same way the use of higher-order functions extends functional programming: the *strategy* pattern – where an object relies on another object received as a parameter to implement part of its behaviour. Using the *strategy* pattern achieves the highest of the four levels of configurability.

Whenever they don't strictly need them, designs shouldn't include unconfigurable objects. For the purpose of optimization or convenience, however, such objects of lesser configurability can be returned by operations on objects of a higher level of configurability. Again, those transformations of higher-order objects into simpler higher-order ones or even first-order ones are only a classical class of operations from the functional programming that includes, for example, currying.

Moreover, many designs of a lower level of configurability can be factored as the result of operations on a design of a higher level of configurability, as can be demonstrated with some of the previously mentioned examples:

- a lazy list of the Fibonacci numbers is the result of the application of a lazy list generating function to the Fibonacci function
- a seeded PRNG is the result of the application of a PRNG generator to a seed (although in practice most PRNG are not functional, and rely on global mutable state instead; thus seeding merely alters the PRNG's seed instead of returning a new number-generating function)
- a fixed-type range function is the result of the application of a range function generator to a function that converts a primitive container type to another container type

We will now illustrate, with the last example, how a seemingly simple specification can easily be implemented with a set of constraints not mandated by the specification, and how sticking to the bare specification and removing unneeded constraints yields a more adaptable result.

A software design company receives a request from a customer: “We need a function that returns a range of integers, that we want to use to conveniently repeat iterations a number of times.”

With a simple specification like this, managers expect a product to be delivered quickly and hope to sell it to many customers. A lot of money is to be made, and programmer Bob, known for his fast work, is given the spec. A first prototype is available in only a few days. Here is its Common Lisp declaration:

```
(defun make-range (end))
```

After testing it, the customer complains: the range contains all integers from 1 up to *end*, inclusive, but he wanted integers from 0 up to *end* - 1. The function is modified accordingly, retested and successfully sold. The first problem arises when the company, to make more profit out of this project, tries to sell this function to other customers. One of them immediately complains that he'd prefer the range to contain integers from 1 up to *end*...

Decision is made to modify the function's signature and let the user specify the starting and ending integers, inclusive:

```
(defun make-range (start end))
```

But then, some customers want to have ranges going downwards or with only even integers. Fortunately, Bob designs a new interface that satisfies both, with an argument to specify the increment between integers:

```
(defun make-range (start end &optional (step 1))
```

As requests come in and need modifications to the code and interface, management begins to worry. Previous customers are angry because they have to change existing code to use the new versions and thus are unwilling to pay much for them, because migrations already cost them. The initial goal was to be able to sell the product as is, without having to invest more money and time in its development. So much for the easy profit.

Still, new feature requests continue to come in that the current interface is unable to express. For example, some customer needs the range in an array and does not want to have to write boilerplate code to do the conversion. Bob decides to create a second function:

```
(defun make-range (start end &optional (step 1))
```

```
(defun make-range-array (start end &optional (step 1)))
```

But on seeing this, many customers ask for others types to be targetted. Someone even asks for ranges to be generated as lazy lists. When another customer asks for heterogeneous ranges to be possible, Bob is desperate and management is ready to cancel the whole project.

At this moment, programmer Alice decides to step up, and suggests to re-design the range library with higher-order and functional programming in mind. As management is willing to avoid a failure, Alice is allocated a very small budget. In not much more time than for the first prototype to be delivered, a new

one is made with the following interface:

```
(defun make-range (&key (start 0) end (step 1)
                  (convert #'identity)))
```

This new version lets the user specify the starting integer (defaulting to 0), the ending integer or a predicate object responsible to decide when new integers must not be generated, the stepping integer or an object responsible to provide successive integers (defaulting to 1) and an object responsible to convert the list of integers to another container type (defaulting to the *identity* function).

As soon as this version is delivered with documentation, no request ever comes in for a modification of the interface, and customers are able to bend the library to their needs.

4 Using signatures as interface contracts

In its essence, a library is nothing more than a signature – including in particular the signatures of its parameters – and proof obligations on the models of objects provided by the library. We briefly recall elements of the algebraic specifications theory about signatures and models [28].

A signature Σ is a triplet $\langle S, F, t \rangle$ where S is a finite non void set of symbols called *sorts*, F is a set of functional symbols and t is the typing application from F to $S^* \times S$.

To each symbol $f \in F$, t associates its arity $w \in S^*$ and its co-arity $s \in S$. In most specification languages, this typing operation is explicitly noted $f : s_1 s_2 \dots s_n \rightarrow s$ with $w = \{s_1 s_2 \dots s_n\}$. This allows the consideration of the signature as a pair $\langle S, F \rangle$, t being implicit. We then say that F is $(S^* \times S)$ -sorted.

For instance, a CafeOBJ [10] specification of the natural integers in Peano's arithmetics could use the following signature:

```
signature N { [Nat]
              op zero : -> Nat
              op succ : Nat -> Nat }
```

Here, the set of sorts S is only $\{\text{Nat}\}$, F is $\{\text{zero}, \text{succ}\}$ and t associates the types following the colon to each symbol of F . Note that the symbol **zero**, having a void arity, is a constant.

The definition of a signature is only syntactic and thus holds no idea of meaning nor behaviour of the functions, even though the names of the symbols generally give an intuition of their intended use. The matching semantic notion of a signature Σ is a Σ -model, or Σ -algebra.

Given a signature $\Sigma = \langle S, F \rangle$, a Σ -algebra or Σ -model is defined by

- a S -sorted set E , disjoint union of sets E_s , with the condition that for each $s \in S$, $E_s \neq \emptyset$

- an application $\bar{f} : E_{s_1} \times E_{s_2} \times \dots \times E_{s_k} \rightarrow E_s$ for each functional symbol $f : s_1 s_2 \dots s_k \rightarrow s$ in F .

For instance, the classical model of the signature \mathbb{N} would associate \mathbb{N} to the set \mathbf{Nat} , the constant 0 to the symbol `zero` and the function $s : \mathbb{N} \rightarrow \mathbb{N}, x \rightarrow x + 1$ to the symbol `succ`.

A typed language allows (or forces, depending whether the typing is made dynamically or statically) the programmer to express a signature of his program, with the sorts being the base types (*e.g.* `int`, `char`, and so on) and the structures and classes declared by the user. The functional symbols as well as the typing application are expressed by the function prototypes.

The compiler of a typed language ensures that all value affectations respect the typing constraints, that is to say that a variable of type s can only receive values of type s . Weak type constraints allow automatic casting, for instance from integers to floating numbers. Object-oriented programs consider a hierarchy of types: if a class c_1 derives from a class c_2 , then all objects of class c_1 also have the type c_2 .

The semantic aspect of each functional symbol is given under the form of the actual program, that is to say the changes the function makes in the variable affectations. To a given signature, one may associate multiple models, that is to say different implementations.

In the case of a library design, only the models of objects provided by the library itself should be constrained. These models don't even need to be fully defined. Giving properties relevant to their use, like pre- and post-conditions as well as algorithmic complexity in time and space, is typically far enough. The interface of a library, then, merely consists in the signature of its parameters. As the respect of a signature by the parameters is the most minimal constraint needed for the library's code to be able to run, imposing no other constraint gives the library's users the highest flexibility.

However, the traditional understanding of the notions of typing, in the imperative programming communities, is that of class hierarchy: attributing a type to a class consists in precisising from which other class it derives and attributing a type to an object consists in precisising to which class it belongs.

Thus, in practice, library designers tend to stick to the simplest type notation for parameters provided by their programming language and thus usually require them to belong to some class. As they design the library with known use cases in mind, they also usually choose a class with a rich model that closely fits those use cases.

In consequence, a user of the library, when faced with the inadequacy of the parameter objects as provided by the library itself, has no choice but to define its own objects as inheriting from the class mandated by the interface. Doing this without error requires knowledge of some parts of the library's implementation

so as to avoid interference of the inherited and newly created members of the objects. This effectively makes the internal details of the class of parameters part of the interface.

Whereas some languages may better support enforcement of type constraints based only on signatures than others, it is important to note that most object-oriented languages currently in use make it practically possible, even if with minor restrictions.

Dynamically typed languages typically provide no notation for a signature other than the use of its operations, thus leading to a run-time enforcement and error handling of signatures. In Common Lisp, the set of operations of a signature is conventionally referred to as a *protocol*. Many functional languages consist more or less, at their core, in a denotational semantics for a typed λ -calculus and provide a syntax for signatures and a compile-time check that a program is properly typed. For example, signatures are defined in Haskell by *type classes*.

Major statically typed imperative languages, on the other hand, don't make it possible to use an arbitrary object respecting a signature as a parameter in a practical way, if possible at all. But they provide a signature-like construct integrated in their class hierarchy. In C++, this construct is the *abstract class* with neither member variables nor concrete methods, in Java it's the *interface*. The object has to inherit from this construct for the typing system to be able to recognize that it respects the signature.

In fact, C++ also provides a powerful *templates* system that enables to truly enforce signatures, but a C++ compiler doesn't use a typing algorithm as powerful as the classical Hindley-Milner type inference algorithm used by functional languages and typing errors on templates typically yield error messages both horribly long and hard to parse by the developer. This makes templates a technically interesting solution but not a practically usable one. This issue has been acknowledged already by the C++ community and the upcoming revision of the C++ standard, C++0x, will include the notion of *concepts*, which basically denote type signatures.

As a library designer, it is beneficial to leverage those facilities when defining a library's interface. By imposing the minimal set of constraints needed on parameters, the designer gains the following:

- the library is more easily reused in situations not anticipated
- the designer can provide alternative parameter objects with conflicting implementations trade-offs (like space and time optimization or inspectability)
- the parameters and the functions of the library can be modified independently as long as they match the signature

Thus a higher-order library using minimal type constraints on its parameters is not only highly reusable but also more easily maintainable.

5 Case study

In this section, we present a case study to illustrate the difference between classical and adaptable code. We show how our approach already works with a rather complicated case, with the example of I/O (Input/Output) streams. We compare two I/O libraries of different languages, the I/O streams of the C++ Standard Template Library (STL) and the Gray Streams extension of Common Lisp. The former is a rather inflexible interface in practice whereas the latter makes I/O highly extensible in Common Lisp.

Paradoxically, the STL is an overall highly configurable library, that in fact achieves to a very high degree to use a signature-constraint higher-order, and provides convenient first-order objects for the most common cases and sane default parameters for some higher-order arguments. For example, strings are parameterized on their character type and generic containers on their content type, and all containers are parameterized on their allocation strategy.

In fact, the STL mostly already used the notion of type signatures in its early releases, but the notion was only informally defined and the signatures named concepts, for the language itself lacked any way to express these type signatures fully.

The STL also provides higher-order algorithms that can be used to implement classical idioms of the functional programming, the use of closures and anonymous functions (a.k.a. lambda functions) notwithstanding. All such algorithms are defined as templates parameterized by the types of all of their arguments, and their proper typing is checked by the compiler, as far as the typing mechanism of C++ is able to type. The typing of C++ also makes it impossible to use inferred types as return values, which promotes the use of side-effects and in-place modifications of containers objects.

In practice, however, in addition to the inherent limitations of the templates, their complexity and the baroque error messages they tend to produce discourage many developers to use them. Thus the libraries they design typically won't define functions with template arguments but with arguments typed with some class provided by the STL. And because I/O streams are used through infix operators that don't mix very well with pointers, C++ late binding cannot be used practically. Arguments are non polymorphic reference or value-based.

Here are two C++ declarations of an I/O method, the first conforming to a "pure" higher-order typing and the second exhibiting a classical typing:

```
template <typename Stream>
void pretty_print(int indent, Stream& s);
```



```
void pretty_print(int indent, std::ostream& s);
```

To the second one, which is very typical of the average C++ I/O function, providing anything but a bare `ostream` class could compile fine but produce run-time errors. Providing an object of a class derived from `ostream`, for example, would be properly typed as far as C++ is concerned, but may result in a segmentation fault during execution. . . Typical implementations of the STL do not seem to expect anything else than a bare `ostream` to be used as arguments, and are typed as the second function, to the contrary of other parts of the STL. The result is that using other streams than the one already provided by the STL is uncommon and cumbersome; one usually needs to reimplement most of the logic of streams, instead of being able to use the existing logic as implemented in the STL.

On the other hand, Common Lisp is a multiparadigmatic language with functional programming as its foundation, built around an untyped λ -calculus. It is typically dynamically typed, which makes it possible to use as an argument of a function call any object respecting the needed signature. Nonetheless, many Common Lisp implementations include a type inference engine, designed to catch as many typing errors as possible at compile-time.

Common Lisp is also the first object-oriented programming language published by the ANSI. Common Lisp Object System, CLOS, is built around classes and generic functions (GF), which are multiply dispatched lately bound functions. Concrete methods are defined for each GF for a set of possibly typed arguments. For example, the above example of an I/O method would be defined in Common Lisp as follows:

```
(defgeneric pretty-print (indent s))
(defmethod pretty-print ((indent integer) s))
```

A signature as defined by a set of generic functions is conventionally called a *protocol* in the Common Lisp community.

As far as I/O is concerned, such a protocol was examined during the standardization of Common Lisp, albeit not meant to be included in the specification of the language itself. The eponymous Gray streams are objects adhering to this protocol, that constitutes a signature for I/O objects. Gray streams constitute an extension to the language¹, and are widely implemented. As such, it is a *de facto* standard.

¹ Common Lisp's streams are handled not by generic functions and methods but plain functions which, in a typical implementation, doesn't lend itself to extension. To comply to the Gray streams specification, Common Lisp's stream handling functions are implemented on top of the Gray streams generic functions, thereby making streams trivially extensible. Also, the stream predicate function, `streamp`, is defined in the spec as being equivalent to `(lambda (x) (typep x 'stream))`, so the implementation must authorize that a CLOS class inherit from the built-in class `stream`.

As there is no need to specifically type stream arguments in functions or methods and any object can act as a stream when methods are defined for the needed generic functions, most libraries designed to do I/O have chosen to implement their I/O objects as Gray streams. A simple higher-order adapter stream, like a filtering or character counting stream, is trivially written in around fifty lines of code. Network streams or graphical interfaces streams all typically are Gray streams, and can easily be composed with existing higher-order streams, like `broadcast-stream` that broadcasts its output to a list of streams or a reencoding stream to perform character encodings translations transparently. Among the available Common Lisp libraries providing such streams, we can also mention, for example, streams whose output is cut as per HTTP's chunked encoding, streams that provide an endpoint for a SSL encrypted tunnel or streams for gzip compressed data. Of course, such streams can trivially be composed to efficiently send compressed chunked data across a secure channel.

It is interesting to note that, in light of the subject of highly reusable libraries, the library here consists in the I/O primitives of Common Lisp, including a `printf`-like facility, `format`. On these primitives, essential parts of the language are built, like the `read` and `print` functions that in turn constitute a major part of the main interaction facility to a Lisp implementation, the Read-Eval-Print Loop (REPL). In this context, streams themselves are to be considered merely as parameters for the standard library, that just happens, as a necessary convenience, to provide a handful a built-in ones, to access files or in-memory character strings or byte vectors.

The flexibility that Gray streams offer, though, leads to a complete reversal, where the user-supplied streams *are* the libraries...

6 Further works

The method highlighted in this paper has a number of drawbacks that may either hinder its use or acceptance among the many practitioners of the field of computer science for which it is of interest – including language designers, library designers and users. In this section, we quickly study some of these and cite possible solutions that would require further inquiry.

An issue that may seem obvious to many is the performance impact of the higher-order. Surely, a bunch of indirections cannot even try to compete on speed with a carefully hand-crafted implementation. However, we wrote two implementations of the `make-range` function described previously, one according to our principle of higher-order that takes functions as its end and step arguments (it also accepts integers and turn them into the corresponding ending predicate or stepper function).

The results of a limited series of timed run seem to indicate that indirections are not a problem at all. The listed implementations, as compiled and run

with the default settings of our Common Lisp implementation, SBCL, showed a approximate 10% increase in speed for the first-order version. But the same functions with type annotations and the compiler set to optimize execution speed showed a 14% increase in speed for the higher-order compared to its previous runs while the first-order version's speed wasn't measurably improved.

Such figures may not hold for every platform, though. In particular, some embedded hardware may not be able to deal efficiently with the indirections when they are not inlined by the compiler.

As unexpected as these figures may be to programmers not aware of functional programming implementation techniques, it has been demonstrated a long time ago that the degraded performance of first-class procedures is merely an artifact of common compilations techniques, originally designed for imperative programming languages [20]. Efficient techniques are since known, like continuation-passing style, that in fact can even show better performances on some classes of programs.

When hand-crafted implementations still perform better, it would be useful to see how far automated proofs of the semantic equivalence of the hand-crafted first-order implementation with the generic higher-order one can be brought. Improving compilers ability to automatically produce an optimized first-order implementation for any call of the higher-order function would also make important the ability to mechanically and automatically prove the correctness of the optimization.

Another rather obvious issue is the readability of the code. Again, it would be a non-issue for any programmer accustomed to functional programming and higher-order programming in particular. But for uneducated programmers and notably for programmers coming from an imperative-only background, both codes implementing and using higher-order may be very hard to understand. We cannot stress enough how important it is to spread a multiparadigmatic approach of programming in curriculums.

Depending on the intended audience of the code, this make comprehensive documentation of the code and of its known use cases rather critical. This also shows how code reusability isn't a purely technical issue, but also a social one.

The third issue of our approach is particularly important to library designers, as it may be influential in their decision to avoid using an unrestricted higher-order in their code.

As some parameters of a library bear no restrictions on their model, but only on their signature, many important properties of the library may not be possible to assert anymore, like termination or algorithmic complexity. Most properties of the library, in fact, may only be possible to assert conditionnally, as functions of the parameters' properties.

Such conditional properties are already common in functional languages spec-

ifications. For example, the `sort` function of Common Lisp has, among others, the following properties:

- if the *key* and *predicate* always return, then the sorting operation will always terminate
- even if the *predicate* does not really consistently represent a total order of the elements, none will be lost (but the elements will be scrambled in some unpredictable way)

It's interesting that the designer of the library may provide to its users different properties of the library for different properties of the parameters. For example, some function may be usable with parameters that can raise exceptions, but with a slightly different semantics than with parameters that are guaranteed not to raise exceptions. . .

Also, not only libraries may benefit from our higher-order approach. As far as configurability is concerned, object-oriented systems of a higher scale than simple programs may be improved by including higher-order in the design decisions. In particular, IPC-based operating systems and distributed architectures could be made notably more flexible with configurability by policy.

For example, instead of registering a restricted form of authentication token, a user may provide to the authentication service a reference to an arbitrary program of its choice, responsible to handle interaction with a user claiming his identity. Of course, as with libraries, default parameters are provided by the system. In this case, a password verification program provided by the system may be chosen by the user.

7 Conclusion

We have shown that when a library designer uses higher-order and sets no other constraints on the function parameters than their type (that is, when object discrimination is made according to properties rather than another criterion), the resulting library reaches a high degree of reusability. We have also considered the possibility that the library functions may be reimplemented by the user if the library specification includes precise proof obligations. The guidelines were illustrated with some classical examples of data structures and algorithms.

We have seen that this issue is also a programming language design issue, as features of the language itself may interfere and remove most of the benefits to be yield by our approach. We may also note that those benefits may be crucial in the wide use of a language's standard library, impacting its overall usefulness. Without both supporting usable higher-order and signature-based type enforcement and providing a higher-order standard library, a programming

language could fail to promote wide reuse of its own standard library. That is, since facilities of the standard libraries are almost considered parts of the language by programmers, a lack of configurability may prevent a wide use of the language.

We believe that the approach recommended in this article should be widely used by the programming communities, for it represents a new opportunity for computers to really be more practical than traditional tools. Nowadays, computer simulations are preferred to physical simulations because of the possibility of reproducibility and of parameter space exploration at a marginal cost.

The lack of adaptability, yet, induces great time losses in big projects whereas there again, computers could do much more. For instance, when a building architect is told, five years after finishing his construction, that elevator norms have evolved or that he has to add facilities to ensure access to disabled people, he has no other choice than to make great works on his building, blasting walls and roofs only to rebuild them later.

Designing libraries with our guidelines in mind gives computers a new asset: there is no configurable concrete, but a virtual construction can become adaptable when correctly built. We know of this possibility, let us just take advantage of it.

We have seen that most programming languages already provide all the features required to follow our guidelines. We think that the main stumbling block towards a wide use of our approach actually resides in the formation of developers to higher-order and its practical usability.

We believe that many other great results of computer science, such as certified programs or structured and functional programming, are currently misunderstood by most developers. Here lies a great challenge which is, unfortunately, probably beyond the scope of research in computer science.

References

1. *Software engineering: Report of a conference sponsored by the NATO Science Committee*, Garmisch, Germany, October 7–11, 1968.
2. J.-R. Abrial. *The B-book: assigning programs to meanings*. 1996.
3. B. Bardin and C. Thompson. Composable ada software components and the re-export paradigm. *Ada Lett.*, VIII(1):58–79, 1988.
4. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development – Coq’Art: The Calculus of Inductive Constructions*. Springer-Verlag, 2004.
5. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
6. E. Damiani and M. G. Fugini. Design and code reuse based on fuzzy classification of components. *SIGAPP Appl. Comput. Rev.*, 4(2):26–32, 1996.
7. P. Devanbu and B. Frakes. Extracting formal domain models from existing code for generative reuse. *SIGAPP Appl. Comput. Rev.*, 5(1):2–14, 1997.

8. K. Dickey. Scheming with objects. *Computer Language*, October 1992.
9. Glenn S. Fowler, David G. Korn, and Kiem-Phong Vo. Principles for writing reusable libraries. *SIGSOFT Softw. Eng. Notes*, 20(SI):150–159, 1995.
10. K. Futatsugi and R. Diaconescu. CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification. *World Scientific, AMAST Series in Computing*, 6, 1998.
11. Jr. G. L. Steele. Building interpreters by composing monads. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 472–492, 1994.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. *Lecture Notes in Computer Science*, 707:406–431, 1993.
13. S. J. Goldsack, A. A. Holzbacher-Valero, R. Volz, and R. Waldrop. Adapt and ada 9x. *Ada Lett.*, XIV(2):80–92, 1994.
14. Object Management Group. *Common Object Request Broker Architecture: Core Specification*. <http://www.omg.org/docs/formal/04-03-12.pdf>, March 2004.
15. N. Haines, D. Kindred, J. G. Morrisett, S. M. Nettles, and J. M. Wing. Composing first-class transactions. *ACM Trans. Program. Lang. Syst.*, 16(6):1719–1736, 1994.
16. T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, 2005.
17. G. C. Harrison. An automated method of referencing ada reusable code using lil. In *WADAS '87: Proceedings of the Joint Ada conference fifth national conference on Ada technology and fourth Washington Ada Symposium*, pages 1–7, 1987.
18. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
19. B. Kitaoka. Establishing ada repositories for reuse. In *TRI-Ada '89: Proceedings of the conference on Tri-Ada '89*, pages 315–323, 1989.
20. D. A. Kranz. *ORBIT: An Optimizing Compiler for Scheme*. PhD thesis, Yale, 1988.
21. G. T. Leavens and C. Clifton. Multiple concerns in aspect-oriented language design: a language engineering approach to balancing benefits, with examples. In *SPLAT '07: Proceedings of the 5th workshop on Engineering properties of languages and aspect technologies*, page 6, 2007.
22. M. D. Lubars. Code reusability in the large versus code reusability in the small. *SIGSOFT Softw. Eng. Notes*, 11(1):21–28, 1986.
23. C. Lüth and N. Ghani. Composing monads using coproducts. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 133–144, 2002.
24. J. C. Michell. *Handbook of Theoretical Computer Science*, volume B, chapter 8, pages 365 – 458. 1990.
25. A. Reedy, C. Shotton, E. Yodis, and F. C. Blumberg. Ada reuse within the context of an ada programming support environment. In *WADAS '88: Proceedings of the fifth Washington Ada symposium on Ada*, pages 11–17, 1988.
26. M. B. Rosson and J. M. Carroll. The reuse of uses in smalltalk programming. *ACM Trans. Comput.-Hum. Interact.*, 3(3):219–253, 1996.
27. F. Steimann. The paradoxical success of aspect-oriented programming. In *OOP-SLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 481–497, 2006.
28. M. Wirsing. *Handbook of Theoretical Computer Science*, volume B, chapter 13, pages 675 – 788. 1990.
29. Benito Zychlinski Z. and Mario Palomar A. A software quality assurance program through reusable code. In *SIGDOC '84: Proceedings of the 3rd annual international conference on Systems documentation*, pages 107–113, 1984.