Proceedings of the 3rd European Lisp Symposium

Fundação Calouste Gulbenkian, Lisbon, May 6-7, 2010

C. Rhodes (ed.)

Sponsors

We gratefully acknowledge the support given to the 3rd European Lisp Symposium by the following sponsors:

FUNDAÇÃO
CALOUSTE
GULBENKIAN

FCT Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E ENSINO SUPERIOR

CÂMARA MUNICIPAL DE LISBOA

FUNDAÇÃO
LUSO-AMERICANA

BANCO ESPIRITO SANTO

Ravenbrook

Organization

Programme Committee

Christophe Rhodes, Goldmiths, University of London, UK (chair)

Marco Antoniotti, Università Milano Bicocca, Italy
Giuseppe Attardi, Università di Pisa, Italy
Pascal Costanza, Vrije Universiteit Brussel, Belgium
Irène Anne Durand, Université Bordeaux I, France
Marc Feeley, Université de Montréal, Canada
Ron Garret, Amalgamated Widgets Unlimited, USA
Gregor Kiczales, University of British Columbia, Canada
António Leitão, Technical University of Lisbon, Portugal
Nick Levine, Ravenbrook Ltd, UK
Scott McKay, ITA Software, Inc., USA
Peter Norvig, Google Inc., USA
Kent Pitman, PTC, USA
Christian Queinnec, Université Pierre et Marie Curie, France
Robert Strandh, Université Bordeaux I, France
Didier Verna, EPITA Research and Development Laboratory, France
Barry Wilkes, Citi, UK
Taiichi Yuasa, Kyoto University, Japan

Local Organization

António Leitão, Technical University of Lisbon & INESC-ID, Portugal (chair)

Edgar Gonçalves, Technical University of Lisbon & INESC-ID, Portugal

Contents

Message from the Programme Chair

Welcome to the third European Lisp Symposium.

This annual event is intended to be the principal European meeting for presentation of novel research, education perspectives in all topics related to the Lisp family of languages. With the milestone of the first half-century of Lisp's existence safely past (and what language communities have the luxury of saying that?) thoughts naturally turn towards the next milestone, one year at a time; as demonstrated at the first two events in this series, in Bordeaux in 2008 and Milan last year, and again this year, the ability of Lisp to generate interest, enthusiasm and growth gives grounds for being optimistic about the reports looking back at the first hundred years of Lisp history.

The programme committee received ten submissions for this year's symposium: each submission was reviewed by four members of the committee, with a discussion period to resolve conflicting reviews. At the end of this process, six submissions (five papers and one tutorial proposal) were selected for presentation at the main track of the symposium. As in previous years, we will aim to build on the symposium by proposing a journal special issue, to which we will invite new contributions as well as extended versions of the accepted papers in these proceedings.

I would like to thank the many people who have invested time and effort in making this event a success: the authors of submissions, the reviewers, and in particular the local organizing team under António Leitão, without whose work, attention to detail, and recent experience as Programme Chair this symposium could not have happened.

Christophe Rhodes, London, April 2010

Message from the Organizing Chair

It gives me great pleasure to welcome you to Portugal, to Lisboa, and to the 3rd European Lisp Symposium.

Historically, this region attracted people from all origins: Gallaeci, Lusitanians, Celtici, Cynetes, Phoenicians, Carthaginians, Romans, Vandals, Suevi, Buri, Visigoths, Alans, Arabs, Berbers, Saqaliba, and Jews. I'm confident the 3rd European Lisp Symposium will contribute to make our city attractive to all tribes of Lispers.

The preparation of a symposium is a huge task and it would have been impossible for me to organize the 3rd European Lisp Symposium without the help of a group of dedicated people. My deepest thanks goes to Christophe Rhodes, an understanding companion during the maddest of times; to Edgar Gonçalves, the man behind our Web site; and to Cassilda Martinho and Ana Matias, who tried as best as it was humanly (and legally) possible to isolate me from the bureaucratic requirements of such an endeavor. I also thank the members of the ELS Steering Committee: their suggestions were always helpful. Last but not least, I want to personally thank João Pavão Martins and Ernesto Morgado for all their encouragement and support, both now and over the last twenty-five years.

Finally, I want to express my debt of gratitude for the generous support provided by our sponsors, namely SISCOG, VILT, Clozure, Ravenbrook, LispWorks, Franz Inc., Ravenpack, Fundação Luso-Americana, Fundação para a Ciência e Tecnologia, Fundação Calouste Gulbenkian, Banco Espírito Santo, Instituto Superior Técnico, and Câmara Municipal de Lisboa. They made it possible.

António Leitão, Lisbon, April 2010

Invited contributions

Going Meta
Reflections on Lisp, Past and Future

Kent Pitman
HyperMeta Inc.
http://www.hypermeta.com/

Over a period of several decades, I have had the good fortune to witness and influence the design, evolution, standardization and use of quite a number of dialects of Lisp, including MACLISP, T, Scheme, Zetalisp, Common Lisp, and ISLISP. I will offer reflections, from a personal point of view, about what enduring lessons I have learned through this long involvement.

Both the programming world and the real world it serves have changed a lot in that time. Some issues that faced Lisp in the past no longer matter, while others matter more than ever. I'll assess the state of Lisp today, what challenges it faces, what pitfalls it needs to avoid, and what Lisp's role might and should be in the future of languages, of programming, and of humanity.

Reading the News with Common Lisp

Jason Cornez
RavenPack International
http://www.ravenpack.com/

The financial industry thrives on data: oceans of historical archives and rivers of low-latency, real-time feeds. If you can know more, know sooner, or know differently, then there is the opportunity to exploit this knowledge and make money. Today's automated trading systems consume this data and make unassisted decisions to do just that. But even though almost every trader will tell you that news is an important input into their trading decisions, most automated systems today are completely unaware of the news – some data is missing. What technology is being used to change all this and make news available as analytic data to meet the aggressive demands of the financial industry?

For around seven years now, RavenPack has been using Common Lisp as the core technology to solve problems and create opportunities for the financial industry. We have a revenue-generating business model where we sell News Analytics – factual and sentiment data extracted from unstructured, textual news. In this talk, I'll describe the RavenPack software architecture with special focus on how Lisp plays a critical role in our technology platform, and hopefully in our success. I hope to touch on why we at RavenPack love Lisp, some challenges we face when using Lisp, and perhaps even some principles of successful software engineering.

Tutorial: Parallel Programming in Common Lisp

Pascal Costanza
Software Languages Lab
Vrije Universiteit Brussel
B-1050 Brussels, Belgium

Parallel programming is the wave of the future: It becomes harder and harder to increase the speed of single-core processors, therefore chip vendors have turned to multi-core processors to provide more computing power. However, parallel programming is in principle very hard since it introduces the potential for a combinatorial explosion of the program state space. Therefore, we need different programming models to reduce the complexity induced by concurrency.

Common Lisp implementations have started to provide low-level symmetric multiprocessing (SMP) facilities for current multi-core processors. In this tutorial, we will learn about important parallel programming concepts, what impact concurrency has on our intuitions about program efficiency, what low-level features are provided by current Common Lisp implementations, how they can be used to build high-level concepts, and what concepts Lispers should watch out for in the near future. The tutorial will cover basic concepts such as task parallelism, data parallelism and pipeline models; synchronization primitives ranging from compare-and-swap, over locks and software transactional memory, to mailboxes and barriers; integration with Lisp-specific concepts, such as special variables; and last but not least some rules of thumb for writing parallel programs.

Lots of Languages, Tons of Types

Matthias Felleisen
College of Computer Science
Northeastern University
Boston, MA 02115

Since 1995 my research team (PLT) and I have been working on a language for creating programming languages—small and large. Our code base includes a range of languages, and others contribute additional languages on a regular basis. PLT programmers don't hesitate to pick our lazy dialect to implement one module and to link it to a strict language for another module in the same system. Later they may even migrate one of the modules to the typed variant during some maintenance task.

An expressive macro system is one key to this riches of languages. Starting with the 1986 introduction of hygienic macros, the SCHEME world has worked on turning macros into tools for creating proper abstractions. The first part of my talk will briefly describe this world of modern macros and its key attributes: hygiene, referential transparency, modularity of macros, phase separation, and macro specification.

The second part of my talk will focus on how to equip LISP-like languages with a sound type systems and that will illustrate the second key idea, namely, monitoring the interactions between different languages. Our approach to type systems allows programmers to stick to their favorite LISP idioms. It mostly suffices to annotate functions and structures with type declarations during maintenance work. To ensure the soundness of this information even when higher-order values flow back and forth between typed and untyped modules, module boundaries are automatically equipped with software contracts that enforce type-invariants at all levels.

Mathematical Applications

Verifying monadic second order graph properties with tree automata

Bruno Courcelle
courcell@labri.fr

Irène A. Durand
idurand@labri.fr
LaBRI, CNRS, Université de Bordeaux, Talence, France

Abstract: We address the concrete problem of verifying graph properties expressed in Monadic Second Order (MSO) logic. It is well-known that the model-checking problem for MSO logic on graphs is fixed-parameter tractable (FPT) [Cou09, Chap 6] with respect to tree-width and clique-width. The proof uses tree-decompositions (for tree-width as parameter) and clique-decompositions (for clique-width as parameter), and the construction of a finite tree automaton from an MSO sentence, expressing the property to check. However, this construction may fail because either the intermediate automata are too big even though the final automaton has a reasonable size or the final automaton itself is too big to be constructed:the sizes of automata depend, exponentially in most cases, on the tree-width or the clique-width of the graphs to be verified. We present ideas to overcome these two causes of failure. The first idea is to give a direct construction of the automaton in order to avoid explosion in the intermediate steps of the general algorithm. When the final automaton is still too big, the second idea is to represent the transition function by a function instead of computing explicitly the set of transitions; this entirely solves the space problem. All these ideas have been implemented in Common Lisp.

Key Words: Tree automata, Monadic second order logic, Graphs, Lisp

1 Introduction

It is well-known from [DF99], [FG06],[CMR01] that the model-checking problem for MSO logic on graphs is fixed-parameter tractable (FPT) with respect to tree-width and clique-width (cwd).

The standard proof is to construct a finite bottom-up tree automaton that recognizes a tree (or clique) decomposition of the graph. However, the size of the automaton can become extremely large and cannot be bounded by a fixed elementary function of the size of the formula unless P=NP [FG04]. This makes the problem hard to tackle in practice, because it is just impossible to construct the tree automaton.

Systematic approaches have been proposed for subclasses of MSO formulas with limited quantifications in [KL09]. Our approach is not systematic; we consider specific problems which we want to solve in practice, for large classes of graphs.

In the general algorithm, the combinatorial explosion may occur each time we encounter an alternation of quantifiers which induces a determinization of the current automaton. We want to avoid determinizations as much as possible. Initial ideas to achieve this goal were first presented in [CD10].

We do not capture all MSO graph properties, but we can formalize in this way coloring and partitioning problems to take a few examples. In this article, we only discuss graphs of bounded clique-width, but the ideas work as well for graphs of bounded tree-width, in particular because if a graph has a tree-width $twd \leq k$, it has a clique-width $cwd \leq 2^{k+1}$. There is however an exponential blow-up.

The Autowrite[1] software written in Common Lisp was first designed to check call-by-need properties of term rewriting systems [Dur02]. For this purpose, it implements tree (term) automata. In the first implementation, just the emptiness problem (does the automaton recognizes the empty language) was used and implemented.

In subsequent versions [Dur05], the implementation was developed in order to provide a complete library of operations on term automata. The next natural step is to solve concrete problems using this library and to test the limits of the implementation. Checking graph properties is a perfect challenge for Autowrite.

Given a property expressed by a MSO formula, we have experimented the three following techniques.

1. compute the automaton from the MSO formula and using the general algorithm,
2. compute directly the final automaton,
3. define the automaton with implicit transition function instead of computing its set of transitions.

The first technique is the only one which is completely general in theory. The two first techniques have the advantage that once the final automaton is computed (and minimized), it can be memorized for further use. The minimal automaton obtained in both cases is unique: it depends only on the property and not on its logical description. This can be helpful to verify that the two constructions are correct.

The limits are soon reached using the first technique. The second technique allows to go somewhat further. With the third technique there is almost no more limitation (at least not the same ones) because the whole automaton is never constructed.

In this paper, we do not address the problem of finding terms representing a graph, that is, to find a clique-width decomposition of the graph. In some cases, the graph of interest may come with a "natural decomposition" from which the clique decomposition of bounded clique-width is easy to obtain but for the general case the known algorithms are not practically usable.

To illustrate our approach, we shall stick to a unique example along the paper although we have made experiments with many more graph properties.

Path Property: Let $Path(X_1, X_2)$ be the monadic second-order formula expressing that, for an undirected graph G and sets X_1 and X_2 of vertices this graph, we have $X_1 \subseteq X_2$, $|X_1| = 2$ and there is a path in $G[X_2]$ linking the two vertices of X_1 [2] .

[1] `http://dept-info.labri.fr/~idurand/autowrite/`

[2] To simplify the presentation, we confuse somewhat syntax and semantics. We note in the same way a variable X_i and its values (sets of vertices)

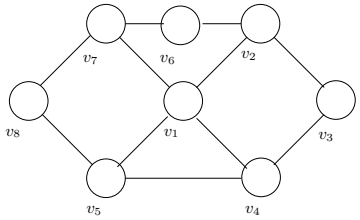


Figure 1: A graph to test the property $Path(X_1, X_2)$

Consider the graph of Figure 1. If $X_1 = \{v_3, v_8\}$ and $X_2 = \{v_1, v_3, v_4, v_7, v_8\}$ the property $Path(X_1, X_2)$ holds for G: $|X_1| = 2$ and there is a path $v_8 - v_7 - v_1 - v_4 - v_3$ from v_8 to v_3 with vertices in X_2. The property does not hold if $X_1 = \{v_3, v_8\}$ and $X_2 = \{v_1, v_3, v_4, v_8\}$.

For $cwd = 2$, we were able to obtain the term automaton (see below how terms describe graphs) directly from the MSO formula starting from the automata representing the basic operations, transforming and combining them with boolean operations, determinization, complementation, projection, cylindrification. But it runs out of memory for $cwd = 3$.

We were successful in constructing the direct automaton for cwd up to 4. But for $cwd = 5$, the program runs out of memory because the constructed automaton is simply too big.

For higher clique-width, there is no way of representing explicitly the transitions. This is when the third method comes on stage. The really new idea here is to represent the transition function precisely by a function. Consequently, there is no more need to store the transitions. Transitions are computed on the fly when the automaton is running on a given term (representing a graph). A graph of clique-width k having n vertices is represented by a term t of size $|t| \leq f(k).n$. Hence, only $|t|$ transitions are needed. This number is in practice much less that the number of transitions of an automaton able to process all possible terms denoting graphs of clique-width $\leq k$.

After recalling how graphs of bounded clique-width are represented by terms and how properties on such graphs can be expressed in MSO, we shall describe our experiments using Autowrite trying to construct automata verifying properties on graphs.

2 Preliminary

2.1 Term automata

We recall some basic definitions concerning terms and term automata. Much more information can be found in the on-line book [CDG$^+$02]. We consider a finite signature

$\mathcal{F}$ (set of symbols with fixed arity) and $\mathcal{T}(\mathcal{F})$ the set of (ground) terms built from a signature $\mathcal{F}$.

Example 1. Let $\mathcal{F}$ be a signature containing the symbols $\{a, b, add_{a\_b}, rel_{a\_b}, rel_{b\_a}, \oplus\}$ with

$$\boxed{\begin{array}{l}\mathsf{arity}(a) = \mathsf{arity}(b) = 0 \quad \mathsf{arity}(\oplus) = 2\\ \mathsf{arity}(add_{a\_b}) = \mathsf{arity}(rel_{a\_b}) = \mathsf{arity}(rel_{b\_a}) = 1\end{array}}$$

We shall see in Section 2.3 that this signature is suitable to write terms representing graphs of clique-width at most 2.

Example 2. t_1, t_2, t_3 and t_4 are terms built with the signature $\mathcal{F}$ of Example 1.

$$\boxed{\begin{array}{l}t_1 = \oplus(a, b)\\ t_2 = add_{a\_b}(\oplus(a, \oplus(a, b)))\\ t_3 = add_{a\_b}(\oplus(add_{a\_b}(\oplus(a, b)), add_{a\_b}(\oplus(a, b))))\\ t_4 = add_{a\_b}(\oplus(a, rel_{a\_b}(add_{a\_b}(\oplus(a, b)))))\end{array}}$$

We shall see in Table 1 their associated graphs.

Definition 1. A (finite bottom-up) *term automaton*[3] is a quadruple $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ consisting of a finite signature $\mathcal{F}$, a finite set Q of states, disjoint from $\mathcal{F}$, a subset $Q_f \subseteq Q$ of final states, and a set of transitions rules Δ. Every transition is of the form $f(q_1, \ldots, q_n) \to q$ with $f \in \mathcal{F}$, $\mathsf{arity}(f) = n$ and $q_1, \ldots, q_n, q \in Q$.

Term automata recognize *regular* term languages[TW68]. The class of regular term languages is closed by the boolean operations (union, intersection, complementation) on languages which have their counterpart on automata. For all details on terms, term languages and term automata, the reader should refer to [CDG$^+$02].

2.2 Graphs as a logical structure

We consider finite, simple, loop-free, undirected graphs (extensions are easy)[4]. Every graph can be identified with the relational structure $\langle \mathcal{V}_G, edg_G \rangle$ where $\mathcal{V}_G$ is the set of vertices and edg_G the binary symmetric relation that describes edges: $edg_G \subseteq \mathcal{V}_G \times \mathcal{V}_G$ and $(x, y) \in edg_G$ if and only if there exists an edge between x and y.

Properties of a graph G can be expressed by sentences of relevant logical languages. For instance, "G is complete" can be expressed by

$$\forall x, \forall y, edg_G(x, y)$$

Monadic Second order Logic is suitable for expressing many graph properties.

[3] Term automata are frequently called tree automata, but it is not a good idea to identify trees, which are particular graphs, with terms.

[4] We consider such graphs for simplicity of the presentation but we can work as well with directed graphs, loops, labeled vertices and edges

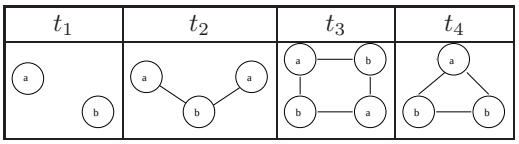


Table 1: Graphs corresponding to the terms of Example 2

2.3 Term representation of graphs of bounded clique-width

Definition 2. Let $\mathcal{L}$ be a finite set of vertex labels and we consider graphs G such that each vertex $v \in \mathcal{V}_G$ has a label $label(v) \in \mathcal{L}$. The operations on graphs are $\oplus$, the union of disjoint graphs, the unary edge addition add_{a_b} that adds the missing edges between every vertex labeled a to every vertex labeled b, the unary relabeling rel_{a_b} that renames a to b (with $a \neq b$ in both cases). A constant term a denotes a graph with a single vertex labeled by a and no edge.

Let $\mathcal{F}_\mathcal{L}$ be the set of these operations and constants.

Every term $t \in \mathcal{T}(\mathcal{F}_\mathcal{L})$ defines a graph $G(t)$ whose vertices are the leaves of the term t. Note that, because of the relabeling operations, the labels of the vertices in the graph $G(t)$ may differ from the ones specified in the leaves of the term.

A graph has *clique-width* at most k if it is defined by some $t \in \mathcal{T}(\mathcal{F}_\mathcal{L})$ with $|\mathcal{L}| \leq k$.

Note also that if the term t describing a graph G does not use redundancies like $add_{a\_b}(add_{a\_b}(\ldots))$, then $|t| = \Theta(|\mathcal{V}_G|)$.

Example 3. For $\mathcal{L} = \{a, b\}$, the corresponding signature has already be presented in Example 1. The graphs corresponding to the terms defined in Example 2 are depicted in Table 1.

Example 4. The graph of Figure 1 is of clique-width ≤ 5. It can be represented with the term built with $\mathcal{L} = \{a, b, c, d, e\}$ and shown on the left of Figure 2.

Let $X_1, \ldots, X_m$ be sets of vertices of a graph G. We can define properties of $(X_1, \ldots, X_m)$. For example,

$E(X_1, X_2)$: there is an edge between some $x_1 \in X_1$ and some $x_2 \in X_2$;
$Sgl(X_2)$: X_2 is a singleton set;
$X_1 \subseteq X_2$: X_1 is a subset of X_2.

Definition 3. Let $P(X_1, \ldots, X_m)$ be a property of sets of vertices $X_1, \ldots, X_m$ graphs G denoted by terms $t \in \mathcal{T}(\mathcal{F}_\mathcal{L})$. Let $\mathcal{F}_\mathcal{L}^m$ be obtained from $\mathcal{F}_\mathcal{L}$ by replacing each constant a by the constants $a\hat{}w$ where $w \in \{0, 1\}^m$. For fixed $\mathcal{L}$, let $L_{P,(X_1,\ldots,X_m),\mathcal{L}}$ be the set of terms t in $\mathcal{T}(\mathcal{F}_\mathcal{L}^m)$ such that $P(X_1, \ldots, X_m)$ is true in $G(t)$, where X_i is

the set of vertices which corresponds to the leaves labeled by $a\hat{}w$ where the i-th bit of w is 1. Hence $t \in \mathcal{T}(\mathcal{F}^m_{\mathcal{L}})$ defines a graph $G(t)$ and an assignment of sets of vertices to the set variables $X_1, ..., X_m$.

Example 5. The graph of Figure 1 with vertex assignment $X_1 = \{v_3, v_8\}$ and $X_2 = \{v_1, v_3, v_4, v_7, v_8\}$ can be represented[5] by the term at the right of Figure 2; it satisfies the path property. With vertex assignment $X_1 = \{v_3, v_8\}$ and $X_2 = \{v_1, v_3, v_4, v_8\}$, it can be represented by almost the same term but with `b^00[v7]` instead of `b^01[v7]` but it does not satisfy the path property anymore.

```
add_c_d(
 add_b_d(
  oplus(
   d[v1],
   rel_d_b(
    add_a_d(
     oplus(
      d[v2],
      add_c_e(
       oplus(
        add_a_b(
         add_b_c(
          oplus(
           a[v3],
           oplus(
            b[v4],
            c[v5])))),
        add_a_b(
         add_b_e(
          oplus(
           a[v6],
           oplus(
            b[v7],
            e[v8])))))))))))
```

```
add_c_d(
 add_b_d(
  oplus(
   d^01[v1],
   rel_d_b(
    add_a_d(
     oplus(
      d^00[v2],
      add_c_e(
       oplus(
        add_a_b(
         add_b_c(
          oplus(
           a^11[v3],
           oplus(
            b^01[v4],
            c^00[v5])))),
        add_a_b(
         add_b_e(
          oplus(
           a^00[v6],
           oplus(
            b^01[v7],
            e^11[v8])))))))))))
```

Figure 2: Terms representing the graph of Figure 1

Example 6. The property $Path(X_1, X_2)$ can be expressed by the following MSO formula:

$\forall x[x \in X_1 \Rightarrow x \in X_2] \wedge$
$\exists x, y[x \in X_1 \wedge y \in X_1 \wedge x \neq y \wedge \forall z(z \in X_1 \Rightarrow x = z \vee y = z) \wedge$
$\forall X_3[x \in X_3 \wedge \forall u, v(u \in X_3 \wedge u \in X_2 \wedge v \in X_2 \wedge edg(u, v) \Rightarrow v \in X_3) \Rightarrow y \in X_3]]$

of quantifier-height 5. Uppercase variables denote sets of vertices, and lowercase variables denote individual vertices.

3 Implementation of term automata

The part of Autowrite which is of interest for this work is the implementation of term automata together with some operations on these automata.

The main operations that are implemented are:

[5] Note that the vertex number inside brackets is not part of the signature; it is there to help the reader make the correspondence between the leaves of the term and the vertices of the graph.

– Reduction (removal of inaccessible states), decision of emptiness; they have been implemented in the very first version of Autowrite.

– Determinization, Complementation, Minimization, Union, Intersection which have been added in subsequent versions of Autowrite.

– Signature transformation, Projection and Cylindrification which have been added to deal with changes of signatures typically from $\mathcal{F}_{\mathcal{L}}^{m}$ to $\mathcal{F}_{\mathcal{L}}^{m'}$.

The object at the core of this library is the term automaton. The efficiency of many operations depends heavily on the data structures chosen to represent the states and transitions of the automata. Since the first version of Autowrite [Dur02], much care has been devoted to improve the representation of automata and the performances have improved significantly. However, this work, which leads us to the limits of what is computable in a human's life, has also shown limits in our implementation, in terms of space and time. In particular, we have realized that representing the set of transitions is a crucial point. SInce, we use binary terms, the number of transitions is $O(s^2)$ where s is the number of states.

From the start, we have represented an automaton as a signed object, (an object with a signature), a list of references to its states, a list of references to its final states and its set of transitions.

3.1 Representation of states

The principle that each state of an automaton is represented by a unique Common Lisp object has been in effect since the beginning of Autowrite. It is then very fast to compare objects: just compare the references. This is achieved using hash-consing techniques. On the contrary to systems like MONA [KM01], a state is not just represented by a number, it can also have constituting elements. The first reason for this choice is that each state has a meaning which can be better expressed by any Lisp object than by a simple number. The second reason is that states can themselves contain states from other automata when building an intersection automaton for example. The third reason will me made clear in Section 6 when we define the transition function as a function instead of defining it as a set of transitions.

Often we need to represent *sets* of states of an automaton. We have two ways of representing sets of states, *bit vectors* or *containers* of ordered states.

Bit vectors are faster, but tend to use more space; containers are slower but can be used when bit vectors lack of space.

Each state has an internal unique number which allows us to order states in the containers. Operations on containers (equality, union, intersection, addition of a state, ...) can then use algorithms on sorted lists which are faster.

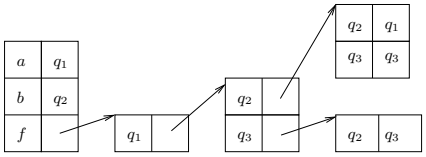


Figure 3: Dag representation of the transitions

3.2 Transitions

The definition itself of an automaton suggests that the transition function should be represented by a set of transitions. And it is indeed the only solution that we had in mind when we started writing Autowrite. Whatever representation is chosen to store the transitions, it must offer a function $\delta(f, states)$ which according to a symbol f of arity n and a list of states $q_1, \ldots, q_n$ returns the target state (or a set of target states in a non-deterministic case) q of the transition $f(q_1, \ldots, q_n) \rightarrow q$ stored in the data structure.

However, the transition function is really a function: if the states have a meaning as specified in Section 3.1, then in some cases, $\delta(f, states)$ can be written as a function which computes the target state q according to f and the contents of the states $q_1, \ldots, q_n$ without the transition being stored in any data structure. We shall explain this novel implementation in Section 6.

The first representation chosen to represent a set of transitions is a hash-table: the key is the list $(f q_1 \ldots q_n)$ (where q_i is in fact the reference to the object representing the state q_i) and the value is the target state q of the transition $f(q_1, \ldots, q_n)$.

For instance, the following set of transitions:

$$\begin{array}{ll} a \rightarrow q_1 & f(q_1, q_2, q_3) \rightarrow q_1 \\ b \rightarrow q_2 & f(q_1, q_3, q_2) \rightarrow q_3 \\ & f(q_1, q_2, q_2) \rightarrow q_2 \end{array}$$

yields a hash-table with 5 entries corresponding to the 5 left-hand-sides of the transitions. The advantage of this representation is that the left-hand-sides are kept together and that we can easily take into account commutative symbols. However, when the symbols have arity $n \geq 2$ the table may become of size $|Q|^n$. In order, to reduce the size of the data structure representing the set of transitions, we have also considered a dag representation which is illustrated by Figure 3.

We now turn our attention to the problem of computing an automaton accepting the terms over $\mathcal{F}_{\mathcal{L}}$ for fixed $\mathcal{L}$ representing graphs verifying an MSO property.

4 The general method (first method)

The first technique consists in applying the general algorithm which transforms a MSO formula into an automaton. The algorithm can be applied recursively until an atomic formula is reached. In order to process a MSO formula, we must translate it into a formula without first-order variables (which has the same quantifier-height) and which uses only boolean operations (and, or, negation) and simple atomic properties like $X = \emptyset$, $Sgl(X)$ (denoting that X is a singleton set), $X_i \subseteq X_j$ for which an automaton is easily computable.

Some standardization on the names of set variables is then necessary in order to apply our operations.

The formula given in Example 6 is thus translated as shown below. Note that this translation is done by hand but could be automated as this is in MONA [KM01].

Example 7.

$$
\begin{aligned}
Path(X_1, X_2) &= X_1 \subseteq X_2 \wedge P_1(X_1, X_2)\\
P_1(X_1, X_2) &= \exists X_3, X_4, P_2(X_1, X_2, X_3, X_4)\\
P_2(X_1, X_2, X_3, X_4) &= Sgl(X_3) \wedge Sgl(X_4) \wedge X_3 \subseteq X_1 \wedge X_4 \subseteq X_1 \wedge X_3 \neq X_4\\
&\quad \wedge |X_1| = 2 \wedge P_4(X_2, X_3, X_4)\\
P_4(X_2, X_3, X_4) &= \neg P_5(X_2, X_3, X_4)\\
P_5(X_2, X_3, X_4) &= \exists X'_1, P_6(X'_1, X_2, X_3, X_4)\\
P_6(X'_1, X_2, X_3, X_4) &= X_3 \subseteq X_5 \wedge \neg X_4 \subseteq X_5 \wedge P_7(X'_1, X_2)\\
P_7(X'_1, X_2) &= \neg P_8(X'_1, X_2)\\
P_8(X'_1, X_2) &= \exists X_3, X_4, P_9(X'_1, X_2, X_3, X_4)\\
P_9(X'_1, X_2, X_3, X_4) &= Sgl(X_3) \wedge Sgl(X_4) \wedge X_3 \subseteq X'_1 \wedge X_3 \subseteq X_2 \wedge X_4 \subseteq X_2 \wedge\\
&\quad Edge(X_3, X_4) \wedge \neg X_4 \subset X'_1
\end{aligned}
$$

4.1 Basic automata for graph properties

We have implemented constructions parametrized by $\mathcal{L}$ of the basic automata which may appear as atomic formulas in our MSO sentences (the leaves of our MSO formulas), among them:

setup-singleton-automaton (cwd m j)	$Sgl(X_j)$
setup-edge-automaton (cwd m i j)	$Edge(X_i, X_j)$
setup-subset-automaton (cwd m j1 j2)	$X_{j_1} \subseteq X_{j_2}$
setup-nequality-automaton (cwd m j1 j2)	$X_{j_1} \neq X_{j_2}$
setup-equality-automaton (cwd m j1 j2)	$X_{j_1} = X_{j_2}$
setup-snequality-automaton (cwd m j1 j2)	$Sgl(X_{j_1}) \wedge Sgl(X_{j_2}) \wedge X_{j_1} \neq X_{j_2}$
setup-cardinality-automaton (cwd m j1 i)	$card(X_{j_1}) = i$

For example, a call to setup-singleton-automaton$(2, 2, 1)$ returns an automaton working on terms representing graphs of clique-width at most 2 (with $\mathcal{L} = \{a, b\}$) with two

```
NAUTOWRITE> (setf *a* (setup-singleton-automaton 2 2 1))
Singleton-X1 2 states  17 rules
NAUTOWRITE> (show *a*)
Automaton Singleton-X1
States q0 q1
Final States q1
Transitions
a^00 -> q0    b^00 -> q0    rel_a_b(q0) -> q0    rel_b_a(q0)  -> q0
a^01 -> q0    b^01 -> q0    rel_a_b(q1) -> q1    rel_b_a(q1)  -> q1
a^10 -> q1    b^10 -> q1    add_a_b(q0) -> q0    oplus(q0,q1) -> q1
a^11 -> q1    b^11 -> q1    add_a_b(q1) -> q1    oplus(q1,q0) -> q1
oplus(q0,q0) -> q0
NIL
NAUTOWRITE> (setf *t* (input-term "add_a_b(oplus(a^10,b^00))"))
add_a_b(oplus(a^10,b^00))
NAUTOWRITE> (recognized-p *t* *a*)
!q1
NAUTOWRITE> (recognized-p *t* *a*)
q1
NAUTOWRITE> (setf *nt* (input-term "add_a_b(oplus(a^10,b^10))"))
add_a_b(oplus(a^10,b^10))
NAUTOWRITE> (recognized-p *nt* *a*)
NIL
```

Table 2: Automaton for $Sgl(X_1)$ with $m = 2$ and $cwd = 2$

sets of vertices X_1 and X_2 and recognizing terms such that X_1 is a singleton, for instance the term `add_a_b(oplus(a^10,b^00))`. An example of such call is shown in Table 2.

4.2 The recursive algorithm

Given a formula $\phi = P(X_1, \ldots, X_m)$, we want to compute the associated automaton $\mathcal{A}(\phi)$.

- If the formula is atomic then we call the function which computes the automaton. For instance, in $P_9(X'_1, X_2, X_3, X_4)$, $Sgl(X_3)$ is computed by

  `setup-singleton-automaton (cwd, 4, 4)`.
- If the formula is a disjunction $\phi = \phi_1 \vee \phi_2$, we compute the union of $\mathcal{A}_{\phi_1}$ and $\mathcal{A}_{\phi_2}$.
- If the formula is a conjunction $\phi = \phi_1 \wedge \phi_2$, we compute the intersection of $\mathcal{A}_{\phi_1}$ and $\mathcal{A}_{\phi_2}$.
- If the formula is a negation $\phi = \neg(\phi')$, we complement the automaton $\mathcal{A}_{\phi'}$. To be complemented $\mathcal{A}_{\phi'}$ must be determinized.
- If the formula is an existential formula of the form $\exists X_j, P(X_1, \ldots, X_m)$, we do a projection of $\mathcal{A}_{P(X_1,\ldots,X_m)}$ on $(1, \ldots, i-1, i+1, m)$ which implies a shift in the indices of variables $X_{i+1}, \ldots X_m$.
- If the formula $\phi = P(X_1, \ldots, X_m)$ does mention X_j, we can obtain $\mathcal{A}_\phi$ by a cylindrification of the automaton $\mathcal{A}_{P(X'_1,\ldots X'_{m-1})}$ (with $X'_i = X_i$ for $1 \leq i < j$ and $X'_i = X_{i+1}$ for $j \leq i < m$) on the j-th components.

Intersection which is handled by saturation (producing a reduced automaton) preserves determinism. The bottleneck of this general algorithm is the necessity of determinizing an automaton in order to complement it. Each determinization can increase exponentially the number of states.

Most properties that we tried could not be tested for graphs of clique-width strictly higher than 2 with this method. It is nevertheless interesting to implement it because it is completely general and for small clique-width we can use the computed automaton for a comparison with the automaton that we obtain using the second method that we are presenting now. The automaton can also be compared with the automaton computed by MONA (see Section 7).

5 The second method: direct construction of the final automaton

The last remark motivates the following development. For some graph properties expressible in MSO, the corresponding automaton can be described directly by a set of states and a description of the transition function on these states. Once a proof has been made that the description is correct (it produces an automaton which recognizes the terms satisfying the property), one can directly compute the automaton without using the MSO sentence. Chapter 6 of the book in progress [Cou09], gives such descriptions for several properties among them $Path(X_1, X_2)$. As said in the introduction, we shall stick to the path property although we can handle many others.

We shall not go into all the details of the construction of the automaton for $Path(X_1, X_2)$, but we shall present at least a description of its states and how the transitions function works.

Let $\alpha(G, x) = \{label_G(y) \mid y \in \mathcal{V}_G \text{ and } x \overset{*}{-}_G y\} \subseteq \mathcal{L}$.
Let $\beta(G) = \{(label_G(x), label_G(y)) \mid x, y \in \mathcal{V}_G \text{ and } x \overset{*}{-}_G y\} \subseteq \mathcal{L} \times \mathcal{L}$.

$$\begin{aligned} Q = \{&Ok, Error\} \cup \{(0, B) \mid B \subseteq \mathcal{L} \times \mathcal{L}\} \cup \\ &\{[1, A, B] \mid \emptyset \neq A \subseteq \mathcal{L}, B \subseteq \mathcal{L} \times \mathcal{L}\} \cup \\ &\{[2, \{A, A'\}, B] \mid A, A' \subseteq \mathcal{L}, A \neq \emptyset, A' \neq \emptyset, B \subseteq \mathcal{L} \times \mathcal{L}\} \end{aligned}$$

The meaning of these states is described in Table 3. We have $2^{cwd^2/2} < |Q| < 2^{cwd^2+2}$ where $cwd = |\mathcal{L}| \geq 2$.

The transition rules are shown in Table 4. In this table, we use the auxiliary functions ($\otimes$, f, g) which can be found in [Cou09].

With the direct construction, we were first able compare the obtained automaton with the automaton obtained with the general method for $cwd = 2$. Then we solved the problem for $cwd \in \{3, 4\}$.

cwd	2	3	4	5
A/min(A)	25 / 12	214 / 127	3443 / 2197	out

State q	Property Pq
$[0, B]$	$X_1 = \emptyset, B = \beta(G(t, X_2)), X_1 = \{v\} \subseteq X_2, A = \alpha(G(t, X_2), v)$
$[1, A, B]$	$B = \beta(G(t, X_2)), X_1 = \{v, v'\} \subseteq X_2, v = v', A = \alpha(G(t, X_2), v),$
$[2, \{A, A'\}, B]$	$A = \alpha(G(t, X_2), d), B = \beta(G(t, X_2))$ there is no path between v and v' in $G(t, X_2)$
Ok	$P(X_1, X_2)$ holds
$Error$	All other cases

Table 3: Meaning of states for the path property $Path(X_1, X_2)$

Transition rules	Conditions
$c\hat{}00 \to [0, \emptyset]$ $c\hat{}00 \to [0, \{(a, a)\}]$ $c\hat{}11 \to [1, \{a\}, \{(a, a)\}]$	$c \in \mathcal{L}$
$rel_{a\_b}(Ok) \to Ok$ $rel_{a\_b}([0, B]) \to [0, h_{a,b}(B)]$ $rel_{a\_b}([1, A, B]) \to [1, h_{a,b}(A), h_{a,b}(B)]$ $rel_{a\_b}([2, \{A, A'\}, B]) \to [2, \{h_{a,b}(A), h_{a,b}(A')\}, h_{a,b}(B)]$	where $h_{a,b}$ replaces a by b
$add_{a\_b}(Ok) \to Ok$ $add_{a\_b}([0, B]) \to [0, B']$ $add_{a\_b}([1, A, B]) \to [1, D, B']$ $add_{a\_b}([2, \{A, A'\}, B]) \to [2, \{D, D'\}, B']$	$B' = f(B, a, b)$ $D = g(A, B, a, b)$ $D' = g(A', B, a, b)$ $(A \odot ((a \otimes b) \circ B)) \cap A' = \emptyset$
$add_{a\_b}([2, \{A, A'\}, B]) \to Ok$	$(A \odot ((a \otimes b) \circ B)) \cap A' \neq \emptyset$
$\oplus(Ok, [0, B]) \to Ok$ $\oplus([0, B], Ok) \to Ok$ $\oplus([0, B], [0, B']) \to [0, B'']$ $\oplus([0, B], [1, A, B']) \to [1, A, B'']$ $\oplus([1, A, B], [0, B']) \to [1, A, B'']$ $\oplus([1, A, B], [1, A', B']) \to [2, \{A, A'\}, B'']$ $\oplus([0, B], [2, \{A, A'\}, B']) \to [2, \{A, A'\}, B'']$ $\oplus([2, \{A, A'\}, B'], [0, B]) \to [2, \{A, A'\}, B'']$	$B'' = B \cup B'$

Table 4: Transition rules of the automaton for $Path(X_1, X_2)$

However, with higher values of clique-width ($cwd \geq 5$), we are confronted to a memory space problem. And indeed the number of states is at least $2^{5^2/2} = 2^{12} \leq |Q|$ which gives at least 2^{25} transitions (see [Cou09], Chapter 6).

We have presented experiments only with the path property. But we have tried several other properties [6] like connectivity, existence of a cycle, k-colorability, ... Most of the time, the limit is around $cwd = 3$. The conclusion is that for greater values of clique-width, it is not possible to compute in extenso the transitions of the automata because its number of states is simply too big (exponential in cwd or more). In a few cases, we do not run out of memory but the program runs "for ever" (3-colorability with $cwd = 3$).

6 The third method: fly-automata

The problems of space (for most properties) or time (coloring property) disappear if we represent transitions with a function. Defining such transitions (which we call *fly-transitions*) consists in defining a lisp function which applies to a symbol f and a list of states $(q_1, \ldots, q_n)$ and returns the target state q of the transitions $f(q_1, \ldots, q_n) \to q$.

This is easily done from the description of the direct construction of the automaton as the one given in Section 5. Actually, the code that is written to define a concrete transition can be directly called in the fly-transitions function.

States that will be accessed when running the automaton on a particular term are initially not known. In most cases, we do not even want to compute the list of accessible states of the automaton because, this list is simply too big to be computed. The states are formally described in a compact way; the ones that are useless will never be computed. The situation is the same for the list of final states. The easiest way to represent final states is also to use a predicate which tells whether a state is final or not.

So a fly-automaton is just a signed object which has a transition function and a final state predicate. Of course Common Lisp is very suitable to represent objects containing functions since functions are first-class objects. Defining a fly-automaton reduces to defining the transition function and final state predicate.

```
(defun fly-path-automaton (cwd)
  (make-fly-automaton-automaton
   (setup-vbits-signature cwd 2)
   (lambda (root states)
     (make-state
      (path-transitions-fun root (mapcar #'state-contents states))))
   (lambda (state)
     (and (ok-p (state-contents state)) state))
   :name (format nil "~A-PATH-X1-X2-fly-automaton" cwd)))
```

The transition function of union and intersection automata is an anonymous function which calls the respective functions of the combined automata. Note that a concrete automaton can be transformed into a fly automaton: the transition function simply looks

[6] See some results at
`http://dept-info.labri.fr/~idurand/autowrite/Graphs/Graphs.pdf`

for the transition in the stored transitions. But the converse may fail for space and time reasons. We did not reach any limitation using fly-automata which we tried up to $cwd = 18$. We could run the automata on terms representing terms on any graph we had a term representation for. Our problem right now is to find big graphs with their clique-decomposition in order to perform tests.

In this paper we did not address the difficult problem of finding a clique-width decomposition of a graph (so the clique-width) of a graph.

This problem was shown to be NP-complete in [FRRS06]. [Oum08] gives polynomial approximated solutions to solve this problem. More can be found in [Cou09].

Often, when automata are used (in compilation for instance), the automaton is "small" and the input is much much larger. In the present case, it is the opposite. In particular, because we do not know how to decompose very large graphs, we are only in position of using our tools for relatively small graphs (say 100 vertices). Consequently, there is no overhead in using fly-automata. Also, it is not important that the terms representing graphs be optimal because the computation "on the fly" of transitions does not depend much on the total number ($|\mathcal{L}|$) of vertex labels.

7 Related work

Monadic second-order logic on finite and infinite words and binary terms is implemented in the software MONA [KM01] developed by Klarlund and others. Its use for checking graph properties is considered by Soguet in [Sog08]. MONA, with some technical adaptations, is usable for the first technique: it is able to automatically compute the automaton corresponding to an MSO formula; in that it seems quicker than Autowrite. States are represented by an integer. MONA works with binary terms only which is ok for graphs represented with a signature with a maximum arity of 2 ($\oplus$). The symbols with higher arity are simply transformed into binary symbols which have fake children when used in terms. The transitions are represented by a two dimensional array. The cell (i, j) contains a binary decision diagram (BDD) which leads for every symbol $f\hat{}w$ to the target state k such that $f\hat{}w(i, j) \to k$. MONA has deterministic transitions only. When a projection is performed, the determinization is done at the same time. Autowrite can deal with symbols of any fixed arity. An important point is that Autowrite has both deterministic and non deterministic automata. This is very useful when the deterministic automaton corresponding to the desired property cannot be computed by lack of space. In that case, Autowrite will be able to check the property with the non deterministic automaton. See also [Cou09] about this last point.

8 Perspectives

We have still many more properties of graph to experiment among them connectivity. For the automata for which we could compute the set of transitions, it would be nice

to create an on-line library of automata corresponding to properties available to the community of researchers. There is still a lot to be done for improving the efficiency of Autowrite. We have maintained several data structures for representing the automata transitions but have not yet conducted systematic tests to evaluate their performances. In order to do more experiments with our fly-automata, we are currently working on a program for generating automatically random or particular graphs (with their decompositions) of arbitrary clique-width.

References

[CD10] Bruno Courcelle and Irène Durand. Tractable constructions of finite automata from monadic second-order formula. In *Workshop on Logical Approaches to Barriers in Computing and Complexity*, Greifswald, Germany, February 2010.

[CDG+02] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 2002. Draft, available from `http://tata.gforge.inria.fr`.

[CMR01] B. Courcelle, J. A. Makowsky, and U. Rotics. On the fixed parameter complexity of graph enumeration problems definable in monadic second-order logic. *Discrete Appl. Math.*, 108(1):23–52, 2001.

[Cou09] Bruno Courcelle. Graph structure and monadic second-order logic. Available at `http://www.labri.fr/perso/courcell/Book/CourGGBook.pdf` To be published by Cambridge University Press, 2009.

[DF99] Rod G. Downey and Michael R. Fellows, editors. *Parameterized Complexity*. Springer-verlag, 1999.

[Dur02] Irène Durand. Autowrite: A tool for checking properties of term rewriting systems. In *Proceedings of the 13th International Conference on Rewriting Techniques and Applications*, volume 2378 of *Lecture Notes in Computer Science*, pages 371–375, Copenhagen, 2002. Springer-Verlag.

[Dur05] Irène Durand. Autowrite: A tool for term rewrite systems and tree automata. *Electronics Notes in Theorical Computer Science*, 124:29–49, 2005.

[FG04] Markus Frick and Martin Grohe. The complexity of first-order and monadic second-order logic revisited. *Annals of Pure and Applied Logic*, 130:3–31, 2004.

[FG06] J. Flum and M. Grohe. *Parameterized Complexity Theory (Texts in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[FRRS06] M. Fellows, F. Rosamond, U. Rotics, and S. Szeider. Clique-width minimization is np-hard. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing*, pages 354–362, Seattle, 2006.

[KL09] Joachim Kneis and Alexander Langer. A practical approach to Courcelle's theorem. *Electron. Notes Theor. Comput. Sci.*, 251:65–81, 2009.

[KM01] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, Aarhus University, January 2001. Notes Series NS-01-1. Available from `http://www.brics.dk/mona/`. Revision of BRICS NS-98-3.

[Oum08] Sang-Il Oum. Approximating rank-width and clique-width quickly. *ACM Trans. Algorithms*, 5(1):1–20, 2008.

[Sog08] David Soguet. Génération automatique d'algorithmes linéaires. Doctoral dissertation (in French), Spcialit: Informatique, July 2008.

[TW68] J.W. Thatcher and J.B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2:57–81, 1968.

A DSEL for Computational Category Theory

Aleksandar M. Bakić
a_bakic@yahoo.com

Abstract Computational category theory is a branch of computer science devoted to the study of algorithms which can be stated in terms of category theory concepts. We describe a language embedded in Common Lisp, CL-CAT, that allows for succinct expression of such algorithms. Code snippets and examples are presented to help bring this abstract topic closer to practice.
Key Words: Lisp, category theory, domain-specific languages

1 Introduction

Category theory is a relatively young branch of mathematics that has been increasingly used in computer science as a modeling tool [4]. Computational category theory (CCT), on the other hand, is a branch of computer science devoted to the study of algorithms which can be stated in terms of category theory concepts [1]. However, category theory already has applications beyond computer science [11, 12], hence the motivation for using CCT in other domains.

A domain-specific embedded language (DSEL) allows to express algorithms of a particular domain more clearly than a general-purpose language, at the same time reusing the infrastructure and tools of the host language [5]. In Lisp, macros and higher-order functions are the most frequently used means for building DSELs [6]. Some DSELs are called libraries or frameworks despite their use to express computation, which is the purpose of a programming language.

In this paper, we describe a CCT DSEL embedded in Common Lisp, named CL-CAT, that has been designed around algorithms presented in [1]. Thanks to its lending support to mixing object-oriented and functional programming paradigms, CLOS [2, 7] is the main ingredient of CL-CAT: class inheritances and precedences are used for CCT entities, and method combinations and multi-methods simplify the expression of certain function computations.

The next section mentions similar software known to us. The description of the design and implementation, and examples of CL-CAT follow in Sections 3 and 4. Basic knowledge of Common Lisp, CLOS and category theory is required. The exposition is informal, hoping to strike the intuition of a software engineer. We close with hindsight remarks.

2 Related Work

The CCT implementation of [1] uses SML, a general-purpose functional language, and is backed by informal proofs. CL-CAT builds upon its approach and emphasizes the practical side, conceptual integrity and reuse.[1] A survey of formalizations of category theory, such as in proof assistants, is given in [3].

Specware^{TM} is a commercial tool that supports the modular construction of formal specifications and their refinement into executable code, based on category theory, sheaf theory, algebraic specification and general logics. It has been developed [13] on top of a CCT layer.

In [14], it is shown how Haskell monads can be composed by computing the coproduct of term representations of their computational effects. This is an elaborate CCT algorithm based on term algebras, the interleaving of induced term *layers*, and quotients of data types. Its approach resembles that of the witness-based unification-as-coequalizer algorithm of [1].

3 Design and Implementation

We give in the sequel the code snippets from the CL-CAT implementation to help explain the most important parts of the design.

3.1 Basic Protocol

In mathematical texts, a *category* is traditionally defined [8, 4] as a structure comprising (i) a collection of *objects*, (ii) a collection of *arrows*, (iii) operations *source* and *target* that each assign an object to an arrow, (iv) operation *identity* that assigns an arrow to an object for which both its source and target are that object, and (v) operation *composition* that assigns to a pair of arrows (g, f), such that the $target(f)$ *equals* $source(g)$, their composite arrow gf, such that $source(gf) = source(f)$ and $target(gf) = target(g)$. The axioms, which must be satisfied by all the objects and arrows of the category, are (1) *identity/unit law*, that the composition of an arrow with the identity arrow of its source or target *equals* the former arrow, (2) *associative law*, that the order of two compositions does not matter, i.e., the resulting arrows are *equal*.

In practice, the "=" used above means identity/sameness, whereas the meaning of "equals" has to be worked out: it may also be the identity, but its context is often the computation/creation of *new* entities.

In CL-CAT, categories are defined by methods of four generic functions, named after the operations defined above:

[1] It should be noted that some code in [1] expressly avoids genericity due to overhead, even though it is considered desirable in general.

```
(defgeneric source (arrow cat))
(defgeneric target (arrow cat))
(defgeneric id (obj cat))
(defgeneric compo (arrow-g arrow-f cat))
```

The identity and associative laws are not enforced but assumed. The `cat` parameter is introduced in each generic function in order to "free" the other parameters from fictitious dependence on it; this design decision reduces the need for copying and supports nicely some CCT algorithms. Whenever unclear in the sequel, keep in mind that it is a complete and consistent set of methods for the above protocol that defines a particular category, not only the `cat` parameter.

Generic object and arrow classes contain slots for read-only contents, `cobj` and `carr`, respectively. The types of these contents are defined for each particular category; for example, in the **FinSet** category of finite sets, `cobj` contains a finite set and `carr` a total function between two finite sets.

```
(defclass obj ()
  ((cobj :initarg :cobj :reader cobj)))
(defun make-obj (c-des cobj)
  (make-instance c-des :cobj cobj))
(defclass arrow ()
  ((carr :initarg :carr :reader carr)))
(defun make-arrow (c-des so to carr cat &rest keys &key &allow-other-keys)
  (add-arrow (apply #'make-instance c-des :carr carr keys) so to cat))
```

Arrow constructor `make-arrow` associates each arrow with a given category object, for bookkeeping purposes.[2] This is the `cat` parameter in the basic functional protocol. An arrow may be associated with more than one category via function `add-arrow`. There are `:around` methods for `source` and `target` that use this bookkeeping information if present.

In some computations, object and/or arrow equality may be needed[3] to implement the meaning of the above "equals" relations:

```
(defgeneric equalc (x y cat))
```

which has the following predefined methods and can further be specialized for a particular category as intensional or extensional. On composite arrows sharing the source and the target, `equalc` coincides with path commutativity.

```
(defmethod equalc :around ((x obj) (y obj) (c cat))
  (or (eq x y) (eq (cobj x) (cobj y)) (call-next-method)))
(defmethod equalc :around ((f arrow) (g arrow) (c cat))
  (and (equalc (source f c) (source g c) c)
       (equalc (target f c) (target g c) c)
       (or (eq f g) (eq (carr f) (carr g)) (call-next-method))))
(defmethod equalc (x y c) nil)
```

It is also useful to abstract out the arrow application:

[2] It could be optimized wrt. `c-des` but for the moment, we aim at flexibility.

[3] It has to be defined and decidable, which may be a problem for objects. Equality should not be confused with isomorphism.

```
(defgeneric arrow-app (a e c))
(defgeneric arrow-range (a c))
(defgeneric arrow-map (a c))
```

The first generic function computes the *point* of the target of `a` that corresponds to `e` in `c`, the second computes the set of such points for the whole source of `a`, and the third returns the set of pairs of such points.

Finally, we add duality, in the arrow-reversing sense, to the basic protocol:

```
(defgeneric dual (x))
```

which assumes that all necessary information is reachable through the single parameter `x`. There is an `:around` method for `dual` that memoizes the relation between dual entities. The first primary method of `dual` is for categories. Their class contains bookkeeping slots, some of which will be explained in the following examples. Macro `with-dual-restart` invokes `call-next-method` and if it fails,[4] it evaluates its body instead, to obtain the result via duality. The elided parts below represent the definition of the remaining methods and slots of `c` and `d`.

```
(defclass cat ()
  ((obj-type :initarg :obj-type :reader obj-type)
   (arrow-type :initarg :arrow-type :reader arrow-type)
   (objs :initarg :objs :accessor objs)
   (arrs :initarg :arrs :accessor arrs)
   (arht :accessor arht :initform (make-hash-table :test #'eq))
   (dual-type :initarg :dual-type :accessor dual-type)))
(defun make-cat (c-des obj-type arrow-type &key
                 (objs nil supplied-objs-p)
                 (arrs nil supplied-arrs-p)
                 (dual-type nil supplied-dual-type-p))
  (let ((c (make-instance c-des :obj-type obj-type :arrow-type arrow-type)))
    (when supplied-objs-p (setf (objs c) objs))
    (when supplied-arrs-p (setf (arrs c) arrs))
    (when supplied-dual-type-p (setf (dual-type c) dual-type))
    c))
(defmethod dual ((c cat))
  (let ((d (make-cat (dual-type c) (obj-type c) (arrow-type c))))
    (defmethod source :around ((a arrow) (cat (eql d)))
      (with-dual-restart (target a c)))
    (defmethod source :around ((a arrow) (cat (eql c)))
      (with-dual-restart (target a d)))
    ...
    (defmethod compo ((g arrow) (f arrow) (cat (eql d)))
      (compo f g c))
    ...
    d))
```

It is important to note that, for example, while it is possible to compose two arrows in a category dual to that in which they are created, it is an error to call `arrow-range` on the resulting arrow having the dual category object as an argument. This is because the arrow is flipped but the `carr` cannot be dualized.

[4] This is done using default primary methods that signal a condition handled by the macro.

3.2 Basic Computations

An entity X, such as an object or an arrow with some structure, computed in CCT may have the *universal property*, or *universality*, meaning that for every other relevant entity, a proof can be constructed that X is distinguished (relative to all these entities). This is implemented via a mixin class `universality` and a generic function `construct`.

```
(defclass universality ()
  ((univ :initarg :univ :accessor univ)))
(defgeneric construct (u &rest args))
(defmethod construct ((u universality) &rest args)
  (apply (univ u) args))
```

A simple example is the initial object:

```
(defclass initial-obj (obj universality) ())
```

for which a method of generic function `compute-initial-obj` binds the slot `univ` of an `initial-obj` X to a function such that `(construct` X O`)` creates a unique arrow from X to O. Thus for all objects O in the same category as X, even for another initial object, this arrow is a proof that X is initial.

To describe a more interesting example, the colimit, we need to introduce the *functor* first. A functor maps each object in category C_1 to an object in category C_2, and each arrow in C_1 to an arrow in C_2, such that these maps commute with all related arrows in C_1 and C_2. The functors can be thought of as arrows in the category of small categories **Cat**, represented by instance `*cat-cat*` of class `cat-cat`. An as example of the change of perspective, frequent in category theory, the source and target of a functor are instances of a `cat-obj` subclass, whose `cobj` contains an instance of a `cat` subclass.

```
(defclass cat-obj (obj) ())
(defclass functor (arrow) ())
(defclass cat-cat (cat) ())
(defparameter *cat-cat* (make-cat 'cat-cat 'cat-obj 'functor))
```

The `carr` of a functor is thus implemented as a cons of two functions: one for objects and the other for arrows; it is up to the designer to ensure that all the commutativity requirements are satisfied. We introduce `functor-app` and its basic methods.

```
(defgeneric functor-app (f o/a))
(defmethod functor-app ((f functor) (o obj))
  (funcall (car (carr f)) o))
(defmethod functor-app ((f functor) (a arrow))
  (funcall (cdr (carr f)) a))
(defmethod compo ((g functor) (f functor) (c cat-cat))
  (make-arrow
   'functor (source f c) (target g c)
   (cons (lambda (o) (functor-app g (functor-app f o)))
         (lambda (a) (functor-app g (functor-app f a))))
   c))
```

A diagram is defined as a functor from an *index category* which, composed with another functor, gives another diagram. It is implemented as a subclass of `functor`, `diagram`, and has a `compo` method similar to that for functors. The index category, implemented using class `abs-cat`, can be thought of as a graph, with nodes as labeled objects and edges as labeled arrows, kept in the `objs` and `arrs` slots, respectively. Algorithms which work directly with index categories, such as colimit object and free algebra computations, traverse these collections.

Next, we need to introduce the *cocone*, an object with additional structure, analogous to the upper bound: a family of arrows having it as their target in C and commuting with all related arrows in C. The mental picture of these arrows and related objects is an index category with a diagram mapping it to the cocone object "over" C. This already complex situation could be further generalized, but by optimization, we only implement the picture as a diagram stored in slot `base` of an `obj` subclass. Slot `sides` contains a function that maps the source of an arrow in the index category to the corresponding arrow in C. The inherited slot `cobj` has the same type as that of any other object in C. The `carr` of a `cocone-arrow`, however, is an arrow in C.

```
(defclass conic-obj (obj)
  ((base :initarg :base :reader base)
   (sides :initarg :sides :reader sides)))
(defclass conic-arrow (arrow) ())
(defun side (c n) (funcall (sides c) n))
(defclass cocone (conic-obj) ())
(defun make-cocone (coapex base sides)
  (make-instance
    'cocone :cobj (cobj coapex) :base base :sides sides))
(defclass cocone-arrow (conic-arrow) ())
```

The colimit object is a generalization of entities such as the initial object in a *cocomplete* category C.[5] Analogous to a least upper bound, it is computed as a "universal completion" over cocone objects, and its class is `colimit-cocone`.[6]

```
(defclass colimit-cocone (cocone universality) ())
(defun make-colimit-cocone (cocone univ)
  (make-instance
    'colimit-cocone :cobj (cobj cocone) :base (base cocone)
    :sides (sides cocone) :univ univ))
```

The `functor-app` methods specialized on the `conic-obj` and `conicarrow`, together with methods on `compo` and `dual`, implement (co)limit-preserving (co)-continuous functors, inheriting the basic behavior via `call-next-method`.

For computing universal completions, two layers of functional protocols are:

```
(defgeneric compute-colimit (d c))
(defgeneric compute-limit (d c))
```

[5] A cocomplete category is one that has colimits on all diagrams. In CCT, this means that they can be computed.

[6] Notice that `univ` parameter of `make-colimit-cocone` pertains to `cocone` parameter, whereas we create a new object to avoid side-effects. Even though `equalc` returns true on the two objects, this is a potential issue.

```
(defgeneric compute-initial-obj (o c))
(defgeneric compute-coproduct (lo ro c))
(defgeneric compute-coequalizer (f g c))
...
```

When the lower-layer methods can be implemented by reusing existing code, going in the other direction is allowed. For example, the default method of `compute-coproduct` below uses `compute-colimit`. The most general method of `compute-colimit` is shown first, using `compute-colimit*`, which in turn uses the lower protocol layer for base cases of a structural recursion. The class names contain `io` when a `compute-initial-obj` method is defined, `cp` when `compute-coproduct` is defined, etc. By theorems, an `io-cp-ce` category inherits from mixin class `cocomplete-cat`, and a `to-pr-eq` category from `complete-cat`.

```
(defmethod compute-colimit ((d diagram) (cat io-cp-ce-cat))
  (compute-colimit* d cat d))
```

Function `cp-diagram` creates a diagram from an index category containing two objects[7] and no arrows, to `c`. The `p-arrow` and `q-arrow` are names used for injections and projections. Instead of `defun`, `def-memoized-fun` memoizes `compute-cp-via-colimit` in order to allow for subdividing client code which depends on the identity: when using the `equalc` is not desired or possible.[8]

```
(defclass coproduct (obj universality)
  ((p-arrow :initarg :p-arrow :reader p-arrow)
   (q-arrow :initarg :q-arrow :reader q-arrow)))
(defmethod compute-coproduct ((lo obj) (ro obj) (c cocomplete-cat))
  (compute-cp-via-colimit lo ro c))
(def-memoized-fun +memo+ compute-cp-via-colimit (a b c)
  (let* ((d (cp-diagram a b c))
         (cc (compute-colimit d c)))
    (make-instance
     'coproduct :cobj (cobj cc)
     :p-arrow (side cc *cp-lo*)
     :q-arrow (side cc *cp-ro*)
     :univ (lambda (o a1 a2)
             (let ((c1 (make-cocone
                        o d
                        (lambda (x)
                          (ecase (cobj x) (cp-lo a1) (cp-ro a2))))))
               (carr (construct cc c1)))))))
```

Using the `dual` method for functors, `pr-diagram` creates a product diagram:

```
(defun pr-diagram (a b c) (dual (cp-diagram a b (dual c))))
```

The category of finite sets, **FinSet**, is bicomplete[9] and in CL-CAT is named `fin-set-cat`. Using the naming convention above, its class hierarchy is:

[7] Since the same index category can be used as the source of all coproduct diagrams, by optimization we use its fixed objects, `*cp-lo*` and `*cp-ro*`, to construct a coproduct object. For diagrams induced by a set of objects and a set of arrows, function `make-diagram` is provided.

[8] The `+memo+` is an instance of key situation, using `equal` for the argument list, from the generic hash-table implementation (see code referred to in [9]).

[9] Both cocomplete and complete, i.e., has all colimits and all limits.

```
(defclass fin-set-cat (cat) ())
(defclass io-cp-ce-fin-set-cat (fin-set-cat io-cp-ce-cat) ())
(defclass to-pr-eq-fin-set-cat (fin-set-cat to-pr-eq-cat) ())
(defclass bicomplete-cat (cocomplete-cat complete-cat) ())
(defclass io-cp-ce-to-pr-eq-fin-set-cat
    (io-cp-ce-fin-set-cat to-pr-eq-fin-set-cat bicomplete-cat) ())
```

Using the method `compute-colimit` on any other `io-cp-ce` category, and method `dual` on a `to-pr-eq` category, method `compute-limit` can be defined to reuse the code for computing a colimit object.

```
(defmethod dual ((c to-pr-eq-cat))
  (let ((dc (call-next-method)))
    (defmethod compute-initial-obj ((o obj) (cat (eql dc)))
      (if *limit-via-dual-colimit*
          (compute-terminal-obj o (dual dc))
          (call-next-method)))
    ...
    dc))
(defmethod compute-limit ((d diagram) (c to-pr-eq-cat))
  (let ((*limit-via-dual-colimit* t))
    (dual (compute-colimit (dual d) (dual c)))))
```

The variable `*limit-via-dual-colimit*` controls the context, in the sense of [10]: the colimit computation is used, but in a context where only limit objects are to be constructed. Methods `compute-pushout` and `compute-pullback` are also defined on cocomplete and complete categories, respectively.

4 Examples

We present in this section three examples, which include computations of gradually increasing complexity.

4.1 Product and Coproduct

For a complete category represented by an instance of class `to-pr-eq-fin-set-cat`, the product of two objects below:

```
(defparameter *fsc-ccat*
  (make-cat 'to-pr-eq-fin-set-cat 'fin-set-obj 'fin-set-arrow))
(defparameter *fsc-a* (make-obj 'fin-set-obj '(a b)))
(defparameter *fsc-b* (make-obj 'fin-set-obj '(c d)))
```

with 2-tuples as conses and sets as lists, pretty-printed looks like:

```
CAT-USER> (compute-product *fsc-a* *fsc-b* *fsc-ccat*)
#<PRODUCT ((A . C) (A . D) (B . C) (B . D))>
CAT-USER> (arrow-app (p-arrow *) '(A . C) *fsc-ccat*)
A
```

This is useful for debugging,[10] even though the above presentation of the `cobj` value does not always match the intuition, which is the canonical Cartesian product. This is because the product object is unique up to isomorphism.

[10] Optionally, for debugging purposes, `print-object` methods can also print unique object identifiers for direct access to CLOS objects that have been printed out.

Similarly, given an instance of class `io-cp-ce-fin-set-cat`:

```
(defparameter *fsc-c2cat*
  (make-cat 'io-cp-ce-fin-set-cat 'fin-set-obj 'fin-set-arrow))
```

the coproduct of the two objects above looks like:

```
CAT-USER> (compute-coproduct *fsc-a* *fsc-b* *fsc-c2cat*)
#<COPRODUCT (^A ^B ~C ~D)>
CAT-USER> (arrow-range (q-arrow *) *fsc-c2cat*)
(~C ~D)
```

where the caret and tilde represents the "left" and the "right" tag of the disjoint union, respectively, consed up with the points but handled by the pretty-printer.

4.2 Relation Composition (Equijoin)

The exposition in [1] is recast in CL-CAT using a mixin `rel-cat` category class, added to the complete **FinSet** category implementation, which partially defines a category whose arrows of class `rel-arrow` contain *span* (multirelation, finite set) objects as `carr`. The composition of two `rel-arrow`s contains a new span that is computed using the pullback (also called fibered product) over the spans of the two `rel-arrow`s; the base case is a pullback of two unique `fin-set-arrow`s to the terminal object (denoted by **1**), which degenerates to a product. In Figure 1, the circular nodes are regular `fin-set-obj` objects, the regular arrows are `fin-set-arrow`s, the rectangular nodes are span objects of the thick `rel-arrow`s; the dotted gray arrows and the universalities are disposed of upon composition.

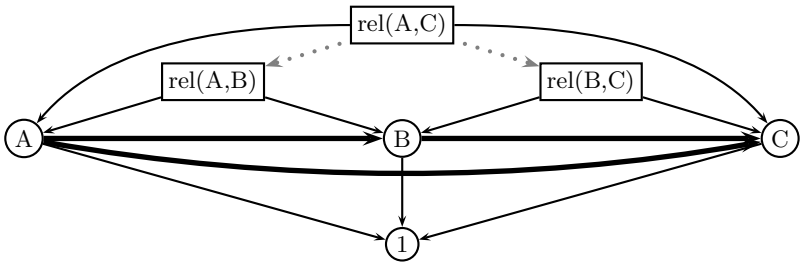


Figure 1: Relation composition using pullbacks, as `rel-arrow` composition

```
(defclass rel-cat (cat) ())
(defclass rel-arrow (arrow) ())
(defclass span (fin-set-obj)
  ((p-arrow :initarg :p-arrow :reader p-arrow)
   (q-arrow :initarg :q-arrow :reader q-arrow)))
(defmethod compo ((g rel-arrow) (f rel-arrow) (c rel-cat))
  (let ((pb (compute-pullback
             (q-arrow (carr f)) (p-arrow (carr g)) c)))
    (make-arrow
     'rel-arrow (source f c) (target g c)
     (make-instance
```

```
      'span :cobj (cobj pb)
      :p-arrow (compo (p-arrow (carr f)) (p-arrow pb) c)
      :q-arrow (compo (q-arrow (carr g)) (q-arrow pb) c))
     c)))
(defun rel-tuples (rel-arrow rel-cat)
  (let ((sp (carr rel-arrow)))
    (remove-duplicates
     (mapcar (lambda (e)
               (cons (arrow-app (p-arrow sp) e rel-cat)
                     (arrow-app (q-arrow sp) e rel-cat)))
             (cobj sp))
     :test 'equal)))
```

The computation of the join of two relations over three `fin-set-obj` objects is then the composition of `rel-arrow`s. Below, pairs are conses and sets are lists.

```
(defclass fsc-cat (rel-cat to-pr-eq-fin-set-cat) ())
(defclass fsc-obj (fin-set-obj) ())
(defclass fsc-arrow (rel-arrow) ())
(defclass fscd-cat (rel-cat io-cp-ce-fin-set-cat) ())
(defparameter *fsc*
  (make-cat 'fsc-cat 'fsc-obj 'fsc-arrow :dual-type 'fscd-cat))
(defparameter *fsc-a* (make-obj 'fsc-obj '(1 2)))
(defparameter *fsc-b* (make-obj 'fsc-obj '(3 4 5)))
(defparameter *fsc-c* (make-obj 'fsc-obj '(6 7)))
(defparameter *fsc-r*
  (let ((o (make-obj 'fsc-obj '((1 . 3) (2 . 3) (2 . 4)))))
    (make-arrow
     'rel-arrow *fsc-a* *fsc-b*
     (make-instance
      'span :cobj (cobj o)
      :p-arrow (make-arrow 'fin-set-arrow o *fsc-a* #'car *fsc*)
      :q-arrow (make-arrow 'fin-set-arrow o *fsc-b* #'cdr *fsc*))
     *fsc*)))
(defparameter *fsc-s*
  (let ((o (make-obj 'fsc-obj '((3 . 6) (4 . 6) (4 . 7) (5 . 6)))))
    (make-arrow
     'rel-arrow *fsc-b* *fsc-c*
     (make-instance
      'span :cobj (cobj o)
      :p-arrow (make-arrow 'fin-set-arrow o *fsc-b* #'car *fsc*)
      :q-arrow (make-arrow 'fin-set-arrow o *fsc-c* #'cdr *fsc*))
     *fsc*)))
```

The class of the category dual to `*fsc*`, `fscd-cat`, is explicitly defined in order to reuse its `compute-colimit` for the pullbacks. The composition is then:

```
CAT-USER> (compo *fsc-s* *fsc-r* *fsc*)
#<REL-ARROW #<SPAN ((3 (1 . 3) (3 . 6) . T) (3 (2 . 3) (3 . 6) . T)
                    (4 (2 . 4) (4 . 6) . T) (4 (2 . 4) (4 . 7) . T))>>
CAT-USER> (rel-tuples * *fsc*)
((1 . 6) (2 . 6) (2 . 7))
```

The `T` comes from products with the terminal object, which is a singleton set {`T`}, because the generic limit computation is used that creates a non-canonical `cobj` value. It is "removed" by the `p-arrow` and the `q-arrow` of the span object, and function `rel-tuples` obtains the set of 2-tuples from it.[11]

[11] Conses and simple Lisp objects are used for `fin-set` elements, the built-in `equal` for element equality, and a function `set-equal` for finite set equality.

It is further possible to compute the intersection `rel-arrow` over a pair of parallel `rel-arrow`s by computing the limit on a diagram induced by the related `fin-set` arrows in `*fsc*`, using the function `make-diagram`. Then similarly, the union `rel-arrow` using the universality of the pushout[12] of `fin-set` arrows from the limit to the pair's `carr`s. Notice how the `compo` multimethod facilitates the implementation of these algorithms, which deal with two categories at the same time, using a single CLOS object, `*fsc*`.

4.3 Three-Valued Logic Topos

We have seen the `fin-set` category, with a minimum of structure. The category of graphs has more structure in the form of imposed constraints such as that an edge has a source and a target. It can be implemented using a parameterized *comma* category, an instance of class `comma-cat`. The parameters are two functors, L and R, that share the target category, and objects in a comma category (L, R) are 3-tuples of objects of the two source categories and a corresponding arrow in the target category; see Figure 2 (left). In the case of finite graphs, the source categories are the same instance of `fin-set`: objects are edge and node sets, respectively; and arrows are functions from the edge sets to (source, target) node-pair sets, both associated with the `fin-set` target category.

A little simpler use of the comma category can help implement the three-valued logic on a topos. First, we describe the implementation of a topos, which is a bicomplete category with an *exponential* and a *subobject classifier*. We do not use an exponential object in the example, so a bicomplete category with a subobject classifier suffices. The subobject classifier Ω, an object with: the logical values, a constant `t-arrow` from the terminal object selecting `T`, and a universality, is defined below.

```
(defclass subobj-classifier (obj universality)
  ((t-arrow :initarg :t-arrow :reader t-arrow)))
(defgeneric truth-carr (cat))
(defgeneric compute-truth-arrow (to omega to-cat))
(defgeneric chi-carr (m cat))
(defgeneric compute-chi-arrow (m omega to-cat))
(def-memoized-fun +memo+ compute-subobj-classifier (to omega to-cat)
  (make-instance
   'subobj-classifier :cobj (cobj omega)
   :t-arrow (compute-truth-arrow to omega to-cat)
   :univ (lambda (m) (compute-chi-arrow m omega to-cat))))
```

The *chi* stands for the characteristic arrow, which maps/classifies a point of the target of a monic arrow `m` to a point in Ω iff it satisfies a criterion of being in the corresponding subobject of the target. It is just a mnemonic for `construct`.

[12] For this, the complete category represented by `fsc-cat` must be extended to a bicomplete category. Note that limits, even in a bicomplete category, must be computed via colimits in their dual category, in order to reuse code.

In the familiar case of `fin-set`, the monic `m` may be a set inclusion (its source is a subset of its target, so it determines a subobject of the target), and each element in the range of `m` is mapped by the characteristic arrow to `T` in Ω. The following function creates an arrow from the source of `m`, `o`, to Ω (`sc`) such that it maps each element of `o` to `T` and, at the same time, commutes with the composition of `m` with its characteristic arrow in `c`.[13]

```
(defun true (o sc c)
  (let ((to (compute-terminal-obj *dummy-obj* c)))
    (compo (t-arrow sc) (construct to o) c)))
```

Methods on the generic `compute-*-arrow` functions above are defined for `fin-set` and `comma-cat`. Methods for the other two are defined on demand.

For the internal logic, we need the terminal object and Ω. Constants are arrows from the terminal object to Ω, unary connectives from Ω to Ω, and binary connectives from $\Omega \times \Omega$ to Ω. For example, as explained in [15], $\vee$ (`top-or`) is defined, given a subobject classifier `sc` and a category `c`, using both colimit and limit computations in `c`. The `epi-mono-factorize` returns a (*mono* `.` *epi*).

```
(def-memoized-fun +memo+ top-or (sc c)
  (let* ((cp (compute-coproduct sc sc c))
         (pr (compute-product sc sc c))
         (st (true sc sc c))
         (si (id sc c))
         (p1 (construct pr sc si st))
         (p2 (construct pr sc st si))
         (m (construct cp pr p1 p2)))
    (chi sc (car (epi-mono-factorize m c)))))
```

Finally, we give the example, then explain further details.

```
(defclass fs-obj (fin-set-obj) ())
(defclass fs-arrow (fin-set-arrow) ())
(defclass fs-cat (io-cp-ce-to-pr-eq-fin-set-cat)
  () (:default-initargs :dual-type 'fsd-cat))
(defclass fsd-cat (io-cp-ce-to-pr-eq-fin-set-cat)
  () (:default-initargs :dual-type 'fs-cat))
(defparameter *fsc* (make-cat 'fs-cat 'fs-obj 'fs-arrow))
(defparameter *l* (make-cocontinuous-functor-via-identity *fsc*))
(defparameter *r* (make-continuous-functor-via-identity *fsc*))
(defclass top-cat (bicomplete-comma-cat)
  () (:default-initargs :dual-type 'topd-cat))
(defclass topd-cat (bicomplete-comma-cat)
  () (:default-initargs :dual-type 'top-cat))
(defparameter *fcc* (make-comma-cat 'top-cat *l* *r*))
(defparameter *src* (make-obj 'fs-obj '(f * t)))
(defparameter *tgt* (make-obj 'fs-obj '(f t)))
(defparameter *to* (compute-terminal-obj *dummy-obj* *fcc*))
(defparameter *omega*
  (make-comma-obj
   *src*
   (make-arrow
    'fs-arrow (functor-app *l* *src*) (functor-app *r* *tgt*)
    (lambda (e) (ecase e (f 'f) (* t) (t t))) *fsc*)
   *tgt*))
(defparameter *sc* (compute-subobj-classifier *to* *omega* *fcc*))
```

[13] The `*dummy-obj*` is used and ignored when we want the terminal object to be unique.

The `*l*` and `*r*` are identity functors that preserve colimits and limits, respectively. This is needed by the `compute-colimit` method on `*fcc*`, which is computed via `compute-colimit` on the source categories. The `compute-limit` method on `*fcc*` uses duality, two more comma categories and an isomorphism.

```
(defmethod compute-limit ((d diagram) (c complete-comma-cat))
  (let* ((idc (iso-dual-comma-cat c))
         (i (make-dual-comma-isomorphism c))
         (*limit-via-dual-colimit* t))
    (limit-via-iso-duality d i idc)))
```

The function `limit-via-iso-duality` implements a generic construction: it first computes the colimit in `idc`, then creates a cocontinuous functor from `i`,[14] uses the two to construct the colimit in the dual of `c`, and finally applies to its copy—which has the `sides` and the arrow created by the universality associated with `c` using `add-arrow`—the `dual` method for colimit cocones. The variable `*limit-via-dual-colimit*` is bound to `T` in order to construct the resulting limit object using limit objects in the (dual of `*fsc*`) `fin-set` category while reusing the colimit computation.

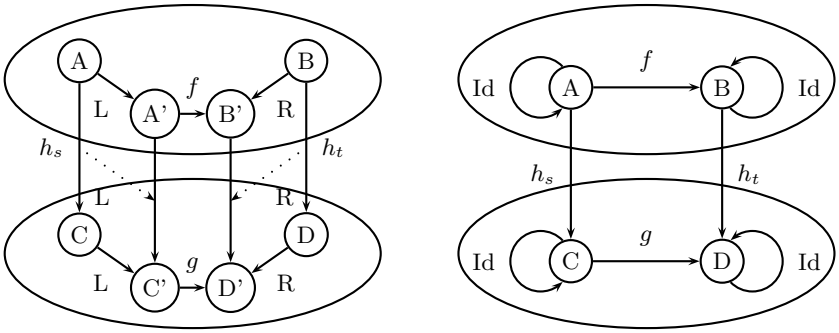


Figure 2: Two comma objects and a comma arrow (h_s, h_t) between them: (left) in general, and (right) for the topos. The rectangles in the middle must commute.

The objects of `*fcc*` are 3-tuples that boil down to just `fin-set` arrows, as shown in Figure 2. The additional structure of this comma category can be seen in its arrows: they are commuting squares which, when Ω is the source and/or target, factor through `*omega*`'s `fin-set` arrow. Therefore, all the constants and the connectives of the topos must be consistent with this function from the 3-valued to the 2-valued set.

Two other parameters of the topos are `truth-carr` and `chi-carr` below. The 3-value part of the `chi-carr` returns `*` for those elements that are in a subset according to the 2-valued logic but not according to the 3-valued logic. The constituent `fin-set` categories of a comma category are kept in a list stored in its `obj-type`, and the two functors in its `arrow-type`.

[14] For `c` being (L, R), `idc` is $(dual(R), dual(L))$ and `i` is an isomorphism between `idc` and the dual of `c`.

```
(defmethod truth-carr ((c top-cat))
  (cons (constant-fn t) (constant-fn t)))
(defun inv-range (a cda cdb c)
  (let ((s (cobj (source a c))))
    (set-difference
     (loop for x in s
        when (member (arrow-app a x c) cdb :test 'equal)
        collect x)
     cda
     :test 'equal)))
(defmethod chi-carr ((m arrow) (c top-cat))
  (let* ((cats (obj-type c))
         (ac (car cats))
         (bc (cadr cats))
         (cc (caddr cats))
         (mc (carr m))
         (mt (target m c))
         (cda (arrow-range (car mc) ac))
         (cdb (arrow-range (cdr mc) bc))
         (cdc (inv-range (cadr (cobj mt)) cda cdb cc)))
    (cons
     (lambda (z)
       (if (member z cda :test 'equal)
           t
           (if (member z cdc :test 'equal) '* 'f)))
     (lambda (z) (if (member z cdb :test 'equal) t 'f)))))
```

Finally, we can compute the 3-valued and 2-valued "truth tables"[15] for the above arrow `top-or`. The method `arrow-map` on a `comma-arrow` returns a cons of the same generic function applied on its constituent arrows.

```
CAT-USER> (arrow-map (top-or *sc* *fcc*) *fcc*)
((((F F . T) . F) ((F * . T) . *) ((F T . T) . T) ((* F . T) . *)
  ((* * . T) . *) ((* T . T) . T) ((T F . T) . T) ((T * . T) . T)
  ((T T . T) . T))
 ((F F . T) . F) ((F T . T) . T) ((T F . T) . T) ((T T . T) . T))
```

5 Conclusion

The work on CL-CAT has been a hobby project with the goal to deepen the understanding of category theory and CCT by coding them in the familiar language. There have been numerous iterations involving extension and refactorization. Whereas the basic CCT is supported at least as much as in [1],[16] without the need for auxiliary data structures such as graphs, many open questions remain in general. The contribution of CL-CAT is mainly a reduced set of CLOS protocols and other constructs that facilitate the experimentation with category-theoretic concepts and CCT algorithms.

Although the concrete syntax is considered important for a DSEL, we have abode by Lisp style guidelines, e.g., not used a macro when a function would

[15] Something similar to the function `rel-tuples` is needed for beautification.

[16] This includes functor and product categories, adjunctions, term algebra, with the connected components, transitive closure and unification algorithms.

suffice. This helps a DSEL to be better embedded in the host language and benefit from existing tools, e.g. for code coverage.

One of the most difficult parts has been the arrow flipping, such as in `limit-via-iso-duality`, related to the approach to duality, which is a novelty of CL-CAT. Further refinement is needed to ensure that duality is propagated in compound objects, even if unused. Equality on objects and points needs more work, perhaps by extending `equalc` and integrating it with the generalized hash-tables. Similarly for the mapping of points, which generalize to diagrams (maps from "map-separating objects" [8]). For the functional CLOS protocols used, only the primary/`:around`/`call-next-method` portion of the standard method combination seem appropriate for the functional nature of CCT.

Acknowledgments. The author is grateful to the anonymous reviewers for their help in improving this work.

References

1. D.E. Rydeheard and R.M. Burstall, *Computational Category Theory*. Prentice Hall International (UK) Ltd., `http://www.cs.manchester.ac.uk/~david/categories/book/book.pdf`, 1988.
2. Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow, CLOS: Integrating Object-Oriented and Functional Programming, Communications of the ACM, v.34, n.9, 1991.
3. Greg O'Keefe, Towards a Readable Formalisation of Category Theory, Proceedings of Computing: The Australasian Theory Symposium (CATS), 2004 .
4. Benjamin C. Pierce, *Basic Category Theory for Computer Scientists*. The MIT Press, 1991.
5. Paul Hudak, Building domain-specific embedded languages, ACM Computing Surveys (CSUR), v.28, n.4es, 1996.
6. Paul Graham, *On Lisp - Advanced Techniques for Common Lisp*. Prentice Hall, 1994.
7. Sonya E. Keene, *Object-Oriented Programming in Common Lisp - A Programmer's Guide to CLOS*. Addison-Wesley, 1989.
8. F. William Lawvere and Stephen H. Schanuel, *Conceptual Mathematics*. Cambridge University Press, 1997.
9. Christophe Rhodes, Robert Strandh, and Brian Mastenbrook, Syntax Analysis in the Climacs Text Editor, International Lisp Conference, 2005.
10. Pascal Costanza and Robert Hirschfeld, Language Constructs for Context-oriented Programming - An Overview of ContextL, Dynamic Languages Symposium, OOPSLA'05, 2005.
11. Thomas Noll, The Topos of Triads, Colloquium on Mathematical Music Theory, 2005.
12. Gerhard Mack, Universal Dynamics, a Unified Theory of Complex Systems. Emergence, Life and Death, Commun.Math.Phys., v.219, 2001.
13. Yellamraju V. Srinivas and James L. McDonald, The Architecture of Specware, a Formal Software Development System, Kestrel Institute, Palo Alto, 1996.
14. Christoph Lüth, Neil Ghani, Composing monads using coproducts, ACM SIGPLAN Notices, v.37, n.9, 2002.
15. Robert Goldblatt, *Topoi: The Categorial Analysis of Logic*, Studies in Logic, v.98, North-Holland Publishing Co., Amsterdam, 1979.

The Outside World

Marrying Common Lisp to Java, and Their Offspring

Jerry Boetje
(College of Charleston, Charleston, SC, USA
boetjeg@cofc.edu)

Steven Melcher
(College of Charleston, Charleston, SC, USA
ssmelche@edisto.cofc.edu)

Abstract: The CLforJava project has devised a number of techniques and patterns to allow seamless access between Java and Common Lisp. These patterns range from meshing type systems to creating `defstruct` architectures to a new view of pathnames. We examine many of these techniques and patterns from the simple to the more complex.

Key Words: Java, Common Lisp, JVM, Intertwining, Architecture

Category: D.2.3 Coding Tools and Techniques

1. Introduction

The open-source CLforJava project is an on-going work to create a new implementation of the Common Lisp specification. The two unique goals of this project are support the education of computer science students and to create a product that intertwines Java and Common Lisp in such a way that users of Java or Lisp view each other as just a library in their language. The first goal continues to be met every year. One half of the latter goal has been attained in that a Java programmer can view Lisp as a Java library. On-going work will extend CLOS in such a way that Java libraries appear as CLOS classes and generic functions.

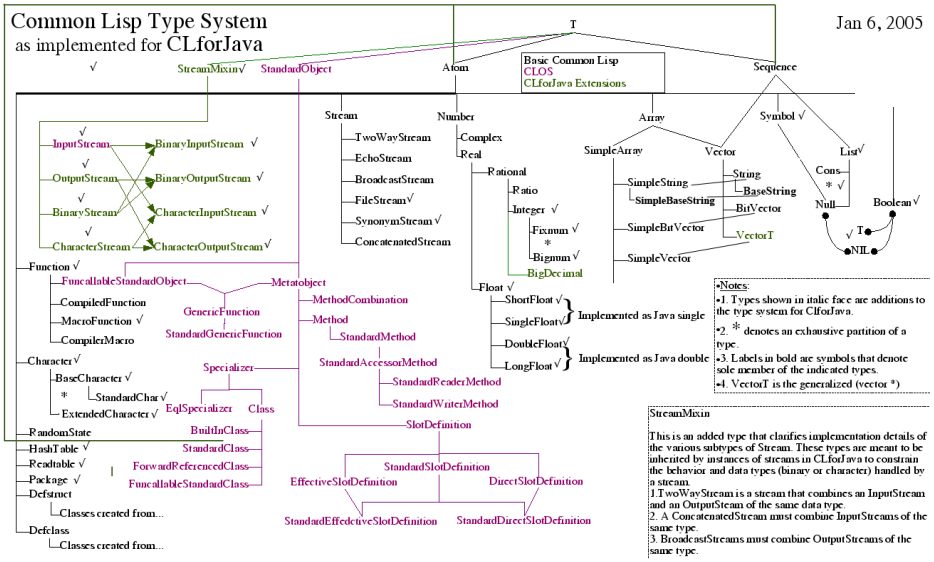
Prior papers have discussed the educational aspect of the project or some of the goals, processes, or a few basic features. This paper details a number of the techniques we have devised to meet these goals.

2.The Obvious

2.1.Meshing the Type Systems

The key to intertwining the two languages is to define one type system in terms of the other's type system. Since CLforJava runs on the JVM, it was logical to define Common Lisp in the Java type system. This diagram depicts the CL type structure includ-

ing those defined in the "The Art of the Metaobject Protocol" (AMOP) as well as types used to mesh the I/O types systems (see the upper left section).



Each of these types is implemented as a Java interface, using Java inheritance to create the type map. Note that the map is not a tree, rather a DAG as specified by Common Lisp. In addition to the Java-based type structure, the interfaces provide several crucial architectural features.

1 They create a bridge between the Java type system and the Common Lisp type system. Each interface contains a reference to the symbol naming the Lisp type. Conversely, the symbol carries a reference to the Java interface defining the type. This example code shows the definition of `ATOM` and that `ATOM` is a subclass of both `BUILT-IN-CLASS` and `T`.

```
package lisp.common.type;
/** Defines an interface for Atom lisp type. */
public interface Atom extends BuiltInClass, T {
    public static final Symbol typeName =
      Package.CommonLisp.intern("ATOM")[0];
```

2. The second feature of this architecture is that any interface that names an instantiable Lisp type carries a static `Factory` class with one or more static `newInstance` methods. For example, to create a Lisp Character given a codePoint, use the following:

```
lisp.common.type.Character.Factory.newInstance(42);
```

3. By defining Lisp types as Java interfaces, any Java program can determine if an object belongs to the Lisp family by using this code fragment:
   `(theObject instanceof lisp.common.type.T)`

2.2.Java Access to Lisp Functions

All Lisp functions are defined by a Java class implementing the Function interface. This is good for Lisp, but how does a Java programmer access a Lisp function? The first step is to define the Java class name of the function by using a declaration - `extensions:java-class-name` - and a string argument that specifies the fully-qualified Java class. For example, to name the `CAR` function, add

```
(declare (ext:java-class-name "lisp.common.function.Car")).
```

There is an additional option when defining a Java-accessible function. If the Lisp function has a non-anonymous name, the implementing class is created with a private constructor preventing instantiation of the class. The compiler however adds a static final field named "`FUNCTION`"and arranges to create a single instance of the class, assigning the instance to that field. From here, any Java code can access that Lisp function via the class and the static field. On the other hand, functions that are not (Lisp) named - effectively anonymous functions - can be treated by Java programmer as any class that can be instantiated. Note that instances of these anonymous functions automatically close over any free variables.

Here is an example of combining the type `CAR` and function `CAR`:

```
lisp.common.type.Cons theCons =
   lisp.common.type.Cons.Factory.newInstance(1, 2);
theCons.setCar(3);
lisp.common.function.Car.FUNCTION.funcall(theCons) => 3
```

Since functions are defined and implemented as classes, closure implementation is straight-forward. A simple, skeleton example illustrates the closure architecture. Every instance of `anon_54` will access the current location of `y` in the instance of `anon_27`. Other instances of `anon_27` will define different locations for `y`.

```
(lambda (x)
  (let ((y x))
    (lambda (z)
      (+ y z))))
```

```
class anon_27 {
  funcall(Object x) {...
    private Object y; ..
    y = x;
    return new class anon_54 {
    funcall(Object z) {
      return (z + y); }
    }}}
```

2.3.Binding Special Variables

CLforJava does not have a single variable binding stack. Each symbol carries its own stack of values as needed. To keep everything sane, the compiler, when it generates binding code, wraps the binding operation in a `try`/`finally` block. There is exactly one `try`/`finally` block for each special variable, preventing unbinding variables that had not yet been bound in this `let`.

```
(let ((*a* 1)
      (*b* 2))
   ...
)
```

```
try {
  *a*.bind(1);
  try {
    *b*.bind(2);
    ...
  } finally {
    *b*.unbind();
  }
} finally {
  *a*.unbind();
}
```

3.Simple But Interesting

3.1.Hash Tables - Implementation and Extension

In Common Lisp, there are four types of hash tables, each with its own equality test for the keys: `eq`, `eql`, `equal`, and `equalp`. However, the Java libraries provide only two types of hash table. Being the type of programmers who would rather use an existing solution than building our own, we investigated the Java solutions. What we found that there was a kernel of a solution, but neither of the Java HashTable nor HashMap could not be used directly.

Here are some differences between Java hash maps and hash tables:

- HashTable is synchronized, and HashMap is not.
  HashMap is better for single threaded applications and can be externally synchronized
- HashTable does not allow null keys or values.
  HashMap allows one null key and any number of null values.

There is one type of hash table common to both Common Lisp and Java. The Java `IdentityHashMap` has the same behavior as defined for the Lisp `EQ` hash table. So that leaves three Lisp types with only one more Java type to try; and of course its behavior is not an exact match for any of them. The behavior of the Java HashMap differs completely from any one of the Lisp ones in that it requires a two-level test of equality. As serendipity has it, the solution to the two-level equality not only lead us to a simple solution for all three of the other hash table types but to a simple mechanism to add new Lisp hash table types nearly trivially.

In Common Lisp, two objects that are equivalent under `eql`, `equal`, or `equalp` may have a different hash code. An example is an array that contains the same characters as a string in Common Lisp. They would both return `T` when tested with `equalp`, but their hash codes may be different. In Java, the hash codes of two keys are compared and must be equal before the keys are tested for equality using the key's Java `equals` method. This would seem to be an insurmountable barrier rather than the solution for three similar problems.

Since we could not rely on the key objects to generate the needed hashCode (telling the `equals` method which algorithm to use - bad idea), we opted for a localized hallucination process whereby we would proffer an altered key to the Java HashMap that will have context information. It is a simple wrapper class that both holds the real key and computes the needed `equals` and `hashCode` methods. There are three subclasses of wrapper to match the three comparisons. It also handles the problem with matching the hash code. The real key is never altered.

Of course, this opens the possibility of adding new types of hash table in CLforJava. This would be an interesting task for one of our students to define a clear interface where the new type could be simply dropped into CLforJava.

3.1.Sorting

Common Lisp provides two sorting functions (simple sort and a stable sort). In CLforJava, they are handled via the Java `Collection sort` method. To use the Java sort, the Lisp `LIST` and `VECTOR` types must also implement the `java.util.Collection` interface. This has the added flexibility for Java programmers who treat the Lisp system as a Java library. However, by using the Java `sort`, the CLforJava `sort` and `stable-sort` functions are identical.

3.2.Starting It All Up

Being a hybrid language processor, the CLforJava startup process is rather unique. Java has a strict set of rules for the loading and initializing classes and interfaces. It also has a penchant for loading classes and initializing classes and interfaces only when needed. As we have found, this on-demand class loading doesn't build a workable system, forcing us to take on more direct ordering of class and interface loading and initialization. Here is the basic class loading order:

1. Make an instance of `lisp.common.CLforJava` - the main class
2. That instance loads (using `Class.forName`)
    1. `lisp.common.type.Package` - all for the standard packages
    2. `lisp.common.type.SpecialOperator` - all of the special operator symbols

3. `lisp.common.type.Declaration` - the standard declaration symbols
4. `lisp.common.type.Variable` - all of the defined variables and their default values
5. `lisp.common.type.Constant` - all of the defined constants and their values

3. The next two operations are designed to force the Java class loader to load and initialize all of the type interfaces and all of the Common Lisp functions that are implemented in Java. Here are they work:
   1. It is critical that the complete type system be loaded at the same time. Failure to load these interfaces and their nested Factory classes leaves the system in an unusable state. To insure its correct operation, each interface that implements a Lisp type carries a `static boolean` variable (`initialize`). The value of this variable is assigned by recursively accessing the value of the `initialize` variable in its parent (or parents in the case of multiple inheritance). The `lisp.common.type.T` interface, of course, starts with a value of `T`. At the end of this algorithm, the entire type system is initialized. On the way, the algorithm also carries a hitchhiker. As it enters a type interface, it puts the Java name of the interface into the Lisp symbol, thereby connecting the two type systems.
   2. Now that the type system is complete, there remains the loading of the core functions - those implemented in Java. It is not sufficient to let Java dynamically load the functions; many will fail to load in time. Our solution also provided an optimization of accessing functions. There is a class, `CommonLispFunctions`, that has a list of all of the defined Common Lisp functions. This list is actually a set of `public` fields where each field holds an instance of a Common Lisp function class. The compiler is aware of this list and will create code to directly access the function by using just a JVM `getField` instruction. This class is initialized immediately after the type initialization process finishes.

The remaining initialization operations are straightforward. The remaining functions are in compiled Lisp and loaded in the proper order. However, we have left one other interesting aspect of startup to discuss. `NIL` is not a small component of nil effort. You see, `NIL` is critical in the earliest parts of startup - those dealing with the package system and the variables and constants. It is vital to be able to create `T` and `NIL` with very little support. To make things worse, `NIL` is a hybrid: part symbol and part list. And both symbols `T` and `NIL` require as their values themselves.

3.3.Unicode Implementation and Useful Extensions

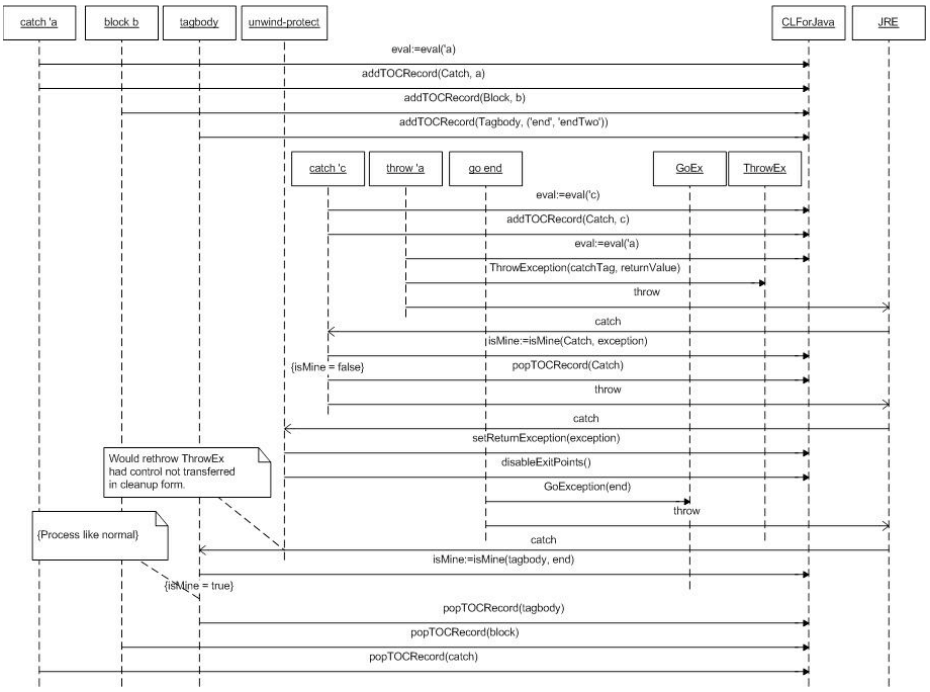
A long-held goal of CLforJava is to handle the full Unicode character set and to provide compatibility with the character and string functions defined in the CL specification. We are please to note that we met that goal with the release of Unicode 4.0 in 2005. During this Fall will update our support to the current Unicode 5.2 specification. Our implementation uses an unorthodox method of dealing with this large set of characters. We chose a brute-force design by creating and loading all 15,000, excluding Korean and Chinese, characters into a hash table on startup. Since all Unicode characters are unique by their number, we can also add other information useful at runtime. Characters are keyed by their number, their Unicode name, Common Lisp name, and other common names such as from the ASCII suite. Other character information is available from some extension functions. A common use of this information is to query for a character's Unicode group. For example, the Reader uses this information when parsing numbers. There are many different sets of the decimal numerals that constitute integers. Our Reader will parse numerals from the same Unicode group and return the integer. We have added an extension to this facility. Unicode does define numerals from non-positional number systems such as Roman or Ancient Aegean. Our current system recognizes Roman numerals (not X,I, etc from ASCII but characters from the Roman Unicode group) and will read (soon write) integers and ratios in Roman numerals.

3.4.Numbers and Arithmetic

CLforJava boxes all numbers to define the Lisp type (fixnum, long float, ratio, etc). All integers are implemented as Java BigIntegers. Most of the numerical functions defined for Common Lisp are also defined in Java, simplifying the implementation. The difficulty arises in the implementation of the binary operations where there are four types of floats, two types of integer, ratios, and complex numbers - leading to a large number of combinations and a large amount of duplicated code. Our approach was to use a Strategy pattern coupled with the Java `enum` facility. The combinations are pre-computed and stored in a set of `EnumMaps`. Given any pair of numbers, it takes no more than 2 accesses (very fast) to the proper method. This method is also amenable to extension to any new number types.

3.5. Transfer of Control (TOC)

A full discussion of TOC in CLforJava is beyond the scope of this paper. In brief, there is a set of Exceptions that carry information about type (catch/go/return-from). A local Java catch is built into all functions. At Lisp exception time, a series of catchers determine the local effects and preform any unwind-protect actions. The following sequence diagram provides an example of a TOC handling in CLforJava. More detailed information is available at
http://clforjava.org/twiki/bin/view/Compiler/TransferOfControl.



An interesting side-effect of this design is that `catch` can also catch Java exceptions by providing the full class name of a Java exception, e.g. `java.lang.Throwable` will catch everything.

4.The Harder Ones

4.1.Filenames and the Network

The Common Lisp pathname specification slightly precedes the ubiquitous presence of the Internet and as a result makes assumptions that apparently to constrict it to a straight-forward implementation that deals with files. However, the specification gives us some leeway in its acknowledge of network-based files by the inclusion of the `HOST` component of a pathname. Our pathnames implementation is based on the Uniform Resource Identifier (URI) and a map of the CL pathname to the URI.

The sharpest turn in the adoption of URIs is that the `DEVICE` pathname component not longer specifies a hardware device such as `C:` but rather specifies a protocol that can access some network-based entity. Under this interpretation, any pathname used access to file-based information will have a `DEVICE` component `:file`. Applying the `OPEN` function to such a pathname acts exactly as the default operation of the Lisp file system. Under this interpretation, the prior use of C: to specify a hard device is now integrated into the `DIRECTORY` as `C|` at the start of the directory form. The `HOST`

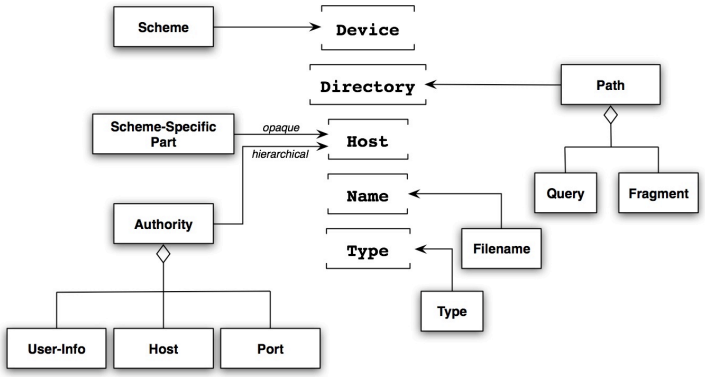
component now becomes the network address/port/authentication . The following table shows 3 possible file pathnames and their components.

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

Under this interpretation, pathnames become powerful entities in Lisp. The `HOST` component carries the instructions needed by the `OPEN` function to make and manage a connection to the network entity. Defining a new pathname type involves creating connection code and specifying the type of operations (`READ-LINE`, `WRITE`, ...) supported. An example built into CLforJava is the use of `HTTP`. Create a pathname with the following elements. Note: the use of 'Scheme' does not refer to the language. It is a term used in networking.

Device:	:http	Name:	index
Host:	www.google.com	Type	html
Directory:	NIL	Version	:unspecific

Open that pathname for `:IO`. Output is handled by sending the name/type components. Read lines until `EOF`. You now have the HTML for Google's first page. The full scope of a CLforJava pathname is quite large. It is possible to build up very powerful pathnames. The following diagram shows the varying options for building from files



and HTTP to handling XML-based and other high-level protocols.

4.2.Compilation Without FASL

CLforJava compiles lisp directly to Java byte code. The `COMPILE` function works entirely in the current environment. Any such compilations do not persist. The COMPILE-FILE function accesses a file of lisp code, compiles it to byte code, and stores it in a file. However, the resulting code is not in a FASL file type. Our system compiles directly to a `JAR` file. Our use of the JAR facility has simplified the loading process, both during startup and later.

Any file compilation produces multiple forms that must be loaded at some later time. Most of the forms will be compiled lambda forms (named `lambda` forms, `LOAD-TIME-VALUE` forms, and other anonymous `lambda` forms). There are also a number of other forms remaining in the file, e.g. `(in-package :foo)`, `(defvar bar 42)`, etc. There is also lisp code that surrounds one or more `lambda` forms from macro expansion. To load these individual forms requires some amount of specialized code, making the compiler larger and buggier. We decided to try a different approach.

Our approach was to compile only lambda forms. In the case of file compilation (and most forms typed into the `REPL`), the complete set of code is wrapped in a lambda form, e.g. `(lambda () ...all-code...)`. This has the effect of adding one more lambda - but a special one. The new lambda is named by the name of the file being compiled. When the compiler encounters another (regular) lambda form, it compiles that lambda and leaves JVM code in the outer form that will make an instance of that lambda class and execute that instance at runtime. This process is recursive such that all lambdas are compiled to classes and instances are created at the proper time.

At the end of compilation, all of the lambdas - including the added one - are written into a standard `JAR` file. The compiler now adds information to the manifest. A critical piece is the `MAIN-CLASS` entry which holds the name of the special file (usually the class name is made from the `JAR` file's name). The `JAR` processing also creates an index of all of the classes that are in the `JAR` file.

At this point, the JAR file is ready to be loaded. The loader locates the JAR file and looks for the `MAIN-CLASS` entry in the manifest. The loader uses a ClassLoader to load that class - the one that wraps everything in the file. The loader accesses the non-argument constructor and executes it. Now that the instance is initialized, the loader calls its `funcall` method. That executes all of the non-lambda code in order and auto-loading the rest of the lambdas referenced in the source file.

4.3. PROVIDE/REQUIRE and the Classpath

This facility is in development and may finished by the symposium. As in other facilities, we look for a Java facility that we can bend to our need. Our implementation is based on the Java class loading process.

As we noted in the compilation section, the `COMPILE-FILE` function creates a special lambda that can be used to locate and execute the startup code for that `JAR` file. The current system creates a class name from the name of the file being compiled, e.g. `quux.lisp`. Using the `PROVIDE` function, the programmer can name the startup class in this file. For example, if the file has the form `(provide "QuuxLisp")`, that would be the name of that special lambda. Of course, it can be anything the programmer wishes provided that the name is valid in Java.

`REQUIRE` uses the same process that LOAD does except that it relies more on the `CLASS-PATH`. If the programmer uses `(require "QuuxLisp")`, `REQUIRE` will ask the current class loader to find that class in its class path. If the programmer gives a list of locations, the function will create a new class loader and set its class path to the list of the locations found in `REQUIRE` and appends the main class path to this new one. Note that this means that can be found in many places since the locations are defined by URIs.

Note also that REQUIRE is recursive. There may a cascade of requests. This will allow us to dispense with our ad-hoc mechanism currently in use.

4.4. LOAD-TIME-VALUE and QUOTE

The implementation of the LOAD-TIME-VALUE special operator gave us an opportunity to put our ability to create the JVM byte-code to the test. The specification requires that the value of the form is evaluated in the NULL environment and during loading into the runtime environment. For CLforJava, that time happens when the Java class implementing the enclosing form is loaded. On detection of an `LTV`, the compiler sets up the actions that will happen during the loading process.

1. Creates a unique name and replaces the `LTV` form with a marker carrying the name.
2. Wraps the `LTV` form in a no-argument `lambda`
3. Compiles the `lambda` form
4. During compilation
    1. adds a static (non-final) field to the form
    2. adds code in class init to make an instance of itself and put it into the static field
    3. makes only a `private` no-argument method
5. Returns to compilation of the enclosing form
6. Adds a static final field in the class
7. Adds code in class init to access the class of the `LTV` form and the static field (the `lambda`)
8. Adds code to set the static field in the `LTV` class to `null`

9. When the compiler encounters the LTV marker, it adds code to access the static field

Here is the sequence of operations occurring while loading of the code of the original, compiled form.

1. Original class init access the static field of the `LTV` class
2. The `LTV` class is initialized which resulted of there being an instance of the `LTV` function
3. Original class calls the `funcall` method of the `LTV` function instance
4. Original class puts that value back into the static final field and now holds the `LTV` value
5. Original class nulls out the static field in the `LTV` form class
6. The `LTV` form class has no other way to be accessed and can be GC'd
7. When code in the original class encounters the LTV marker, it just access the static final field

A bit intricate, but it happens only once, at the correct time (loading in the runtime environment), and leaves no tracks, and access to the value is very fast.

Dealing with quote is so similar to `LTV` that CLforJava does just that. When the compile encounters a `quote` form that is the least bit complex (eg not numbers, characters, strings, or symbols), the form is transformed to an `LTV` and processed accordingly.

4.5.Using Annotations

To date, Java annotations have not played a significant role in CLforJava. Their only use is to hold the documentation string of a function. However, we have built on this annotation to improve the internationalization of the product. When the compiler encounters a doc string, it creates a unique tag and stashes the string into a Java resource keyed by the tag. The tag is also wired into the function class as a static final field. The `documentation` function accesses the class field to get the tag and returns the string the resource.

The location of the resource varies depending on the type of compilation. If the compilation is an in-memory operation, the resource is just attached to the created class. However, if the class is created as the result of a file compilation, the operation is extended. At the end of the compilation, the compiler revisits each of the classes that have the string resource and gathers all of the function doc strings. These are written out to a resource (property file) using the tags as the keys. The `documentation` function first attempts to find the string in the function class. If there is no direct access to the string, it accesses the resource to find the string.

This would not be worth of the time to discuss it except for two additional actions. First, the resource obeys the rules for Java properties including the ability to auto-

matically access a translated (localized) version of the documentation. The user can change the language by setting or binding the `extensions:*locale*` variable. The second characteristic is that the doc string is wrapped in an XML format which includes other information (such as the name of the function and the date/time it was created). That supports the use of XSL transformations to display the documentation in a variety of forms such as plain text or HTML.

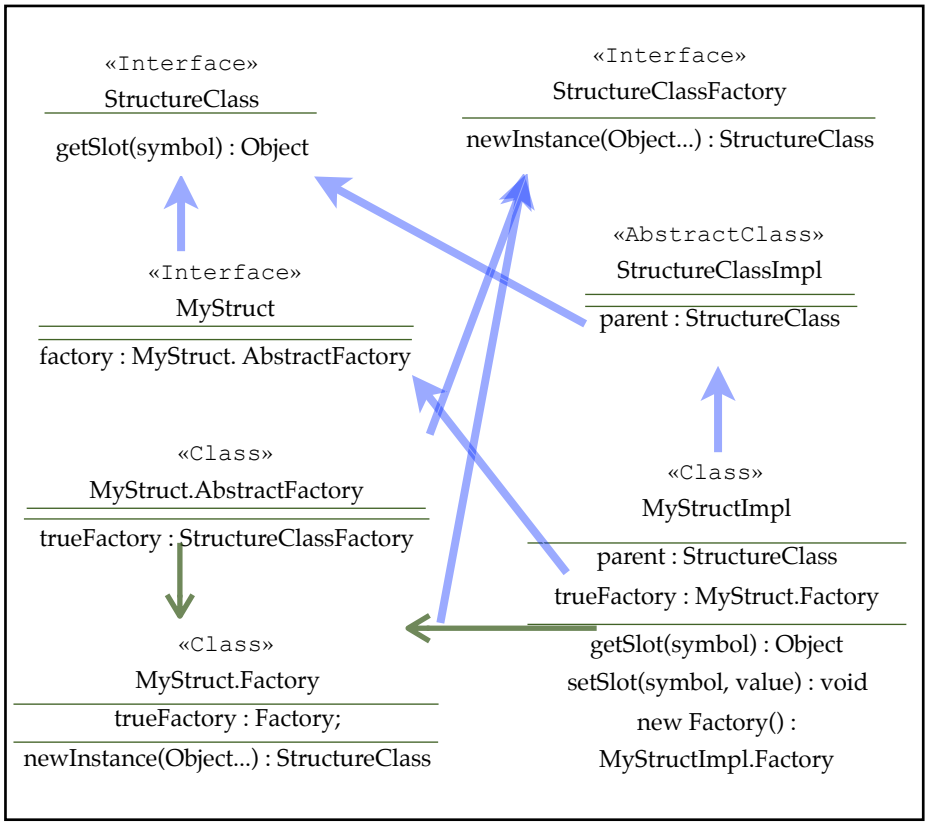
5.Then There's DEFSTRUCT

5.1.Overview

`DEFSTRUCT` is one of the few facilities in Common Lisp that define new types. In addition, it is possible to redefine an existing structure while retaining the type definition - something that is difficult using the Java transformation facility. To make it work in both worlds, we defined a pattern of classes that, in concert, can create new structure classes, alter existing structure classes, make instances of the structure, act as super classes (without having to make an actual subclass inheritance) for `include` options, and retain the ability to access instances of prior structure instances when the structure class is redefined.

The architecture follows the Abstract Factory pattern and is implemented around three components, two interfaces and an abstract class, that define the core of the `defstruct` facility. The interfaces, `StructureClass` and `StructureClassFactory`, define the common components for all structures and factories. The other three components specialize the type of the structure (the type of the `defstruct`), define the concrete factory to create instances of the struct, and the instances of the structure type themselves. While this seems overly complex, but except for the code that manipulates the slots in an instance, the 6 interfaces and classes take no more than 25 lines of Java code.

The following diagram illustrates the structure architecture:



The design also supports reuse of the type-defining Java interfaces when a `defstruct` is redefined. For example, given a structure definition `(defstruct foo a b c)`, instance slots can be accessed by the forms `FOO-A`, `FOO-B`, and `FOO-C`. If the struct were altered as `(defstruct foo d a f)`, the accessors would be `FOO-D`, `FOO-A`, and `FOO-F`. All existing instances of the prior struct can still be accessed by `FOO-A`, `FOO-B`, and `FOO-C`. The `FOO-A` accessor works correctly for both definitions. Note that the type of both struct instances are considered to be type `FOO` in Lisp and `Foo` in Java. A similar mechanism works with included structs. A programmer may specify the Java type of a structure by use the of `extensions:java-class-name` declaration.

6. Conclusions

To date, this attempt to build a new Common Lisp by intertwining it with Java has been a success. Our next large sub-projects are to make a new compiler written entirely Lisp and implement the CLOS facility. The CLOS implementation is our vehicle for directly accessing any Java library from Lisp without requiring Foreign Function Interface (FFI). Work has begun on the new compiler which should be online

next Fall. The new compiler will be more stable and much smarter. The work on CLOS has started with basic data structures and a version of the topological sort that can include Java classes in the sort. The remaining major component is the Condition system and its integration into the Java runtime, although some investigation has started.

7.References

Common Lisp For Java: An Intertwined Implementation, Boetje J, International Lisp Conference, 2005

Unicode 4.0 in Common Lisp: Adoption of Unicode 4.0 in CLforJava, Boetje J, International Lisp Conference, 2005

Foundational Actions: Teaching Software Engineering When Time Is Tight, Boetje J, Proceedings of the 11th Annual SIGCSE Conference on Innovations and Technology In Computer Science Education, 2006

A Metaobject Protocol for CLforJava, Cotton J, Boetje J, Proceedings of the 2007 International Lisp Conference, 2007

Tutorial: Computer Vision with Allegro Common Lisp and the VIGRA Library using VIGRACL

Benjamin Seppke, Leonie Dreschler-Fischer
Department Informatik, University of Hamburg, Germany
{seppke, dreschler}@informatik.uni-hamburg.de

1 Introduction

In this tutorial we present the interoperability between the VIGRA C++ computer vision library and Allegro Common Lisp. The interoperability is achieved by an extension called VIGRACL, which uses a multi-layer architecture. We describe this architecture and present some example usages of the extension. Contrary to other computer vision extensions for Common Lisp we focus on a generic design, which allows for easy interoperability using other programming languages like PLT Scheme.

VIGRA is not just another computer vision library. The name stands for "Vision with Generic Algorithms". Thus, the library that puts its main emphasis on customizable and therefore generic algorithms and data structures (see [Köthe 1999]). It uses template techniques similar to those in the C++ Standard Template Library (STL) (see [Köthe 2000]), which allows for an easy adaption of any VIGRA component to the special needs of computer vision developers without losing speed efficiency (see [Köthe 2010]). The VIGRA library was originally designed and implemented by Ullrich Köthe as a part of his Ph.D. thesis. Meanwhile, many people are involved to improve the library and the user group is growing further. The library is currently in use for various educational and research tasks in German Universities (e.g. Hamburg and Heidelberg) and has proven to be a reliable (unit-tested) testbed for low-level computer vision tasks.

Although C++ can lead to very efficient algorithmic implementations, it is still an imperative low-level programming language, that is not capable of interactive modeling. Thus, the VIGRA library also offers specialized numpy-bindings for the Python programming language as a part of the current development snapshot.

Functional programming languages like Lisp on the other hand provide an interesting view on image processing and computer vision because they support symbolic processing and thus symbolic reasoning at a higher abstraction level. Common Lisp has already proven to be adequate for solving AI problems, because of its advantages over other programming languages, like. the extendability of the language, the steep learning curve, the symbolic processing, and

the clarity of the syntax. Moreover, there are many extensions for Lisp like e.g. description logics, which support the processes of computer vision and image understanding.

2 Related work

Before introducing the VIGRACL interface, we will compare some competitive Common Lisp extensions, which also add image processing capabilities to Common Lisp:

The first system is the well-known OBVIUS (Object-Based Vision and Understanding System) for Lisp (see [Heeger and Simoncelli 2010]). It is an image-processing system based on Common Lisp and CLOS (Common Lisp Object System). The system provides a flexible interactive user interface for working with images, image-sequences, and other pictorially displayable objects. It was last updated on 1994, so it does not supply state-of-the-art algorithms used for image processing and computer vision.

ViLi (Vision Lisp) has been developed by Francisco Javier Snchez Pujadas at the Universitat Autnoma de Barcelona until 2003 (see [Snchez Pujadas 2010]). Although it offers algorithms for basic image processing tasks, it seems to be restricted to run only at Windows and to be no longer maintained.

IMAGO is another image manipulation library for Common Lisp developed by Matthieu Villeneuve until 2007 (see [Villeneuve 2010]). It supports file loading/saving in various formats and image manipulation functionalities . Although it supports some basic filtering and composition tools, there are important image processing parts missing like segmentation tools or the Fourier transform.

The last system is called ch-image (cyrus harmon image) and was last updated in 2008 (see [Harmon 2010]). Like OBVIUS, it provides a CLOS Lisp layer for image processing, but puts its main emphasize on computer graphics (e.g. creating images containing graphical elements). This system introduces many different classes for different image pixel types and therefore requires more studying of the API than systems on a more abstract level.

Contrary to these systems, our proposed VIGRACL binding is generic, lightweight, and offers advanced functions like image segmentation. There is no need for introducing new data types or classes (using CLOS) in Allegro Common Lisp. It currently provides the basic functionalities of the VIGRA library and will be extended in future to 3D processing methods etc. Further, we do not use named parameters, but named the function arguments according to their meaning, which will be shown by auto-completion e.g. when using EMACS in conjunction with Allegro Common Lisp.

Note that we will present a generic interface to the VIGRA library, that allows for the use of many other languages and programming styles, although we favor

the use of the VIGRA in together with functional languages, like Common Lisp or PLT Scheme (see [Seppke 2010]). The generic approach is also reflected in the platform availability, as the VIGRACL has already proven to work with Allegro Common Lisp at Windows, Linux or Mac.

3 Hierarchical Design of the VIGRACL

We will now present the embedding of the VIGRA's algorithms into Allegro Common Lisp. We will start with the lowest layer, and show how an image is represented on the C++ side. We then iteratively move up layer to layer to end by the Common Lisp high-order functions, which assist at the use of the library. As an example, we show the representation of a classical image smoothing filter at each level.

3.1 Lowest Layer: C++

At the lowest layer an image is represented by the VIGRA-Class `BasicImage<T>`. The template parameter `T` determines the pixel-type of the image and can either be a scalar type or a vector type (like [R,G,B]). At this layer we abstract a to allow only two types of images: Images with a floating-point pixel-type and images with a (R,G,B)-Vector of floating-point values. Multi-band images may be supported in future releases.

```
try{
    //Create result image of same size
    vigra::BasicImage<float> img2(img.width(), img.height());

    //Perform filtering with sigma=3.0
    vigra::gaussianSmoothing(srcImageRange(img),
                             destImage(img2), 3.0);
}
catch (vigra::StdException & e){  }
```

3.2 Intermediate Layer: C (shared object)

At the intermediate layer, we create C-wrapper functions around the C++ functions and types of the lowest layer. We favor the use of an own implementation instead of using automatic wrapper generation libraries like SWIG (see [Beazley 1996]) to keep full control of the interface and its definitions. To allow for an easy re-use of all the functions contained inside this shared object, we make some minimalistic assumptions according to the programming language, which connects to this wrapper library:

Image-bands are represented as one-dimensional C-arrays of type float, which is sufficient for most tasks. For matrix-computation one-dimensional C-arrays of

type double are used. Additional function-parameters can either be boolean, integer, float or double. The resulting type of each function is an integer, which is also used to indicate errors. The pointer to these C-arrays will not be created, maintained or deleted by the wrapper library. The client (Allegro Common Lisp) has to take care of all array creation ans deletion tasks. All image functions are yet defined to work band-wise.

It should be mentioned, that this layer results in a shared object (or DLL under Windows) with a very simple interface: All interface signatures consist of elementary C-data types or pointers to float and double. This simplifies the use for other programming languages. Besides Allegro Common Lisp, we have implemented interfaces for PLT Scheme and ittvis IDL (see [Seppke 2010]).

```
LIBEXPORT int vigra_gaussiansmoothing_c( const float *arr,
   const float *arr2, const int width, const int height,
   const float sigma){
   try {
      // create a gray scale image of appropriate size
      vigra::BasicImageView<float> img (arr,  width, height);
      vigra::BasicImageView<float> img2(arr2, width, height);

      vigra::gaussianSmoothing(srcImageRange(img),
                               destImage(img2), sigma);
   }
   catch (vigra::StdException & e) {
      return 1;
   }
   return 0;
}
```

3.3 High layer: Allegro Common Lisp FFI

We now connect the wrapper library to Allegro Common Lisp using the built-in Foreign Function Interface (FFI). For the representation of images we have chosen lists of the (2D) array ype of the FFI. Each array depicts a certain band of an image. The bands are ordered '(Red Green Blue) for color images and '(Gray) for grayscale images. We have chosen Allegro's FFI instead of CFFI because of the native array memory sharing between Common Lisp and C. The corresponding FFI-adapter on the lisp side for the Gaussian smoothing of a single band is given by:

```
(ff:def-foreign-call vigra_gaussiansmoothing_c
   ((arr      (:array :float))
    (arr2     (:array :float))
    (width    :int fixnum)
    (height   :int fixnum)
    (sigma    :float))
 :returning   (:int fixnum))
```

Note that the creation of images is also perfomed on this layer. The Lisp function for the gaussian smoothing of an image band on this layer is given below:

```
(defun gsmooth-band (arr sigma)
  (let* ((width  (array-dimension arr 0))
         (height (array-dimension arr 1))
         (arr2 (make-array (array-dimensions arr)
               :element-type (array-element-type arr)
               :initial-element 0.0)))
    (case (vigra_gaussiansmoothing_c arr arr2
                                     width height sigma)
      ((0) arr2)
      ((1) "Error in vigracl.filters.gsmooth: ..."))))
```

3.4 Highest layer: Allegro Common Lisp High-Order Functions

At the topmost layer, we provide a set of generic and flexible tools, which assist in mapping and to traversing of images. These functions refer to both images and image-bands. They can be seen as extensions to the well-known high-order functions in Common Lisp, but fitted to the data types of images and image-bands.

The first set of functions corresponds to the `mapcar` for lists. We define `array-map`, `array-map!`, `image-map` and `image-map!` for this purpose. We will present their use in the examples section of this tutorial. We also define folding operations for bands and images: `array-reduce` and `image-reduce`. Further, we introduce simple image- and band-iterator functions to write and apply own algorithms to images: `array-for-each-index` and `image-for-each-index`.

Most image processing functions use the mapping facilities to apply a band-defined image operation on each band of an image. For instance the gaussian smoothing of an images is defined of the gaussian smoothing of all image's bands:

```
(defun gsmooth (image sigma)
  (mapcar #'(lambda (arr) (gsmooth-band arr sigma)) image))
```

Note that the VIGRACL introduces its own package namespace (`VIGRACL`) and is defined as a Common Lisp system using Allegro's `excl:defsystem`.

4 Interactive Examples

We will now present some examples that show the practical use of the VIGRACL. The necessary first step is to let Allegro Common Lisp know where the library is located. Afterwards, we can load the library using a single command:

```
CL-USER> :ld vigracl
```

If the library has been loaded successfully, you can switch into the package by typing:

```
CL-USER> (in-package :vigracl)
```

You should get a response message from the Lisp system, which indicates that you are inside the correct package. We now start with loading an image (taken from [Ssawka 2010]):

```
VIGRACL> (setq img (loadimage "images/tools-bits.jpg"))
```

You can easily proof that this image is a RGB image by typing:

```
VIGRACL> (length img)
```

which counts the number of bands of the loaded image (three). Let us now try some basic filters of the VIGRA that are based on convolution with Gaussian kernels: Gaussian smoothing and the calculation of the Gaussian gradient magnitude (both at scale $\sigma = 2.0$):

```
VIGRACL> (setq smooth_img (gsmooth img 2.0))
VIGRACL> (setq gradient_img (ggradient img 2.0))
```

(a) original image (b) smoothed image (c) gradient magnitude

Figure 1: The loaded image and filter results

We now demonstrate how to save these images using the `saveimage` function and review the result using your favourite image browser, or view the result directly using the built-in function `showimage`, i.e. for the smoothed image:

```
VIGRACL> (saveimage smooth_img "images/smooth_bits.png")
VIGRACL> (showimage smooth_img)
```

Both functions will return true, if the saving is successful and print out error messages otherwise. Note that various image formats are supported for the import and export of images.

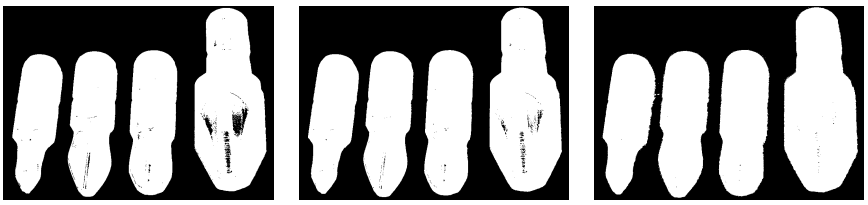
The next task will be counting the bits that are visible on our loaded image. They appear darker that the background on all bands, so we will work with a single-band image from now on:

```
VIGRACL> (setq gray_img (list (second img)))
```

We have arbitrary selected the green channel and defined a new image. To detect whether a pixel corresponds to an object, we will start with writing our first own image processing algorithm, a thresholding filter:

```
VIGRACL> (defun threshold (image val)
           (image-map #'(lambda (x)
               (if (> x val) 0.0 255.0)) image))
VIGRACL> (setq thresh_img (threshold gray_img 220.0))
```

As you can see in Figure 2, there is no threshold that perfectly separates the objects from the background. On a low threshold, many parts inside the bits remain classified as background whereas on the higher threshold some image content around the objects is misclassified.



(a) $t = 190$ (b) $t = 220$ (c) $t = 250$

Figure 2: Results of thresholding the green channel of Figure 1(a)

For a correct segmentation of the bits from the background, we need to close the holes inside of the thresholded image (with $t = 220$). This operation is performed by a morphological operator called closing. We select a radius of 5 pixel to close the unwanted holes:

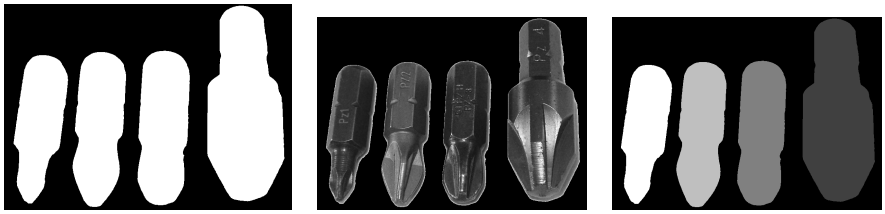
```
(setq closed_img (closingimage thresh_img 5))
```

We can also visualize the segmentation to mask out the original image using the high-order functions of the VIGRACL:

```
(showimage (image-map #'(lambda (i m) (* i (/ m 255.0)))
                      gray_img closed_img))
```

At last, we need to count the to connected components of the image to get the number of objects. This is performed by a labeling algorithm, which assigns each component an unique label. Note that the background will also be counted as one component. Therefore, the result has to be decremented by one.

```
(setq label_img (labelimage closed_img))
(setq bit_count (- (first (image-reduce #'max label_img 0.0))
                   1))
```



(a) segmented image (b) masked green channel (c) labeled image

Figure 3: Results after the closing image operation of Figure 2(b)

This finally results in a `bit_count` of 4, which means that four objects have been recognized (see Figure 3(c)). Next possible steps could be the measurement of size, mean intensity or other features of each labeled component.

5 Conclusions

We have presented some of the functionalities for the VIGRACL extension to Allegro Common Lisp. The extension uses a multi-layer architecture to grant access to the computer vision algorithms that are provided by the VIGRA library. Note that this tutorial cannot be more than an introduction into the interesting field of computer vision besides the presentation of the inter-operational design of the VIGRACL. However, it shows how easy the various functions of the VIGRA can be used within Allegro Common Lisp, given a light-weight generic interface.

We have shown that the integrated high-order functions for images and image-bands help when working with the library as they extend the well-known high-order functions by means of image-bands and images. Thus, we currently use the VIGRACL-bindings for research to assists with the low-level image processing tasks that have to taken out before the symbolic processing. Another advantage is the use for fast interactive image analyzing in scientific computing.

Due to the steep learning curve and interactive experience, we currently use the very similar VIGRAPLT (a VIGRA interface to PLT Scheme using the same intermediate layer) in undergraduate student computer vision projects.

References

[Beazley 1996] D. Beazley, SWIG: an easy to use tool for integrating scripting languages with C and C++, In: Proceedings of the Fourth USENIN Tcl/Tk Workshop, 1996, pp. 129-139.

[Harmon 2010] C. Harmon, The ch-image Homepage, `http://cyrusharmon.org/static/projects/ch-image/doch/ch-image.xhtml` (Jan. 27, 2010)

[Heeger and Simoncelli 2010] D. Heeger, E.Simoncelli, The OBVIUS Homepage, `http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/vision/obvius/0.html` (Jan. 27, 2010)

[Köthe 1999] U. Köthe, Reusable Software in Computer Vision, in: B. Jhne, H. Hauecker, P. Geiler: "Handbook on Computer Vision and Applications", vol. 3, A. Press, 1999.

[Köthe 2000] U. Köthe, STL-Style Generic Programming with Images, in : C++ Report Magazine 12(1), January 2000

[Köthe 2010] U. Köthe, The VIGRA Homepage, `http://hci.iwr.uni-heidelberg.de/vigra/` (Jan. 26, 2010).

[Roerdink and Meijster 2000] J. B. T. M. Roerdink, and A. Meijster (2000). The watershed transform: Definitions, algorithms, and parallellization strategies. In Goutsias, J. and Heijmans, H., editors, Mathematical Morphology, vol. 41, pages 187228. IOS Press.

[Snchez Pujadas 2010] F. J. Snchez Pujadas, The ViLi Homepage, `http://www.cvc.usb.es/~javier/vili.htm` (Jan. 27, 2010)

[Seppke 2010] B. Seppke, Digital image processing using Allegro Common Lisp and VIGRA = VIGRACL, `http://kogs-www.informatik.uni-hamburg.de/~seppke/index.php?page=vigracl&lang=en` (Jan. 26, 2010).

[Ssawka 2010] Ssawka, Image of four bits, `http://commons.wikimedia.org/w/index.php?title=File:PZ_1-4_0.jpg&oldid=26986252`

[Villeneuve 2010] M. Villeneuve, IMAGO Common Lisp image manipulation library, `http://matthieu.villeneuve.free.fr/dev/imago/`, (Feb. 1, 2010)

The Language

CLOX: Common Lisp Objects for XEmacs

Didier Verna
(EPITA Research and Development Laboratory, Paris, France
didier@xemacs.org)

Abstract ***CLOX*** is an ongoing attempt to provide a full Emacs Lisp implementation of the Common Lisp Object System, including its underlying meta-object protocol, for XEmacs. This paper describes the early development stages of this project. ***CLOX*** currently consists in a port of Closette to Emacs Lisp, with some additional features, most notably, a deeper integration between types and classes and a comprehensive test suite. All these aspects are described in the paper, and we also provide a feature comparison with an alternative project called EIEIO.

Key Words: Lisp, Object Orientation, Meta-Object Protocol
Category: D.1.5, D.3.3

1 Introduction

Note: *the author is one of the core maintainers of XEmacs. In this paper, the term Emacs is used in a generic way to denote all flavors of the editor. If a distinction needs to be made, we use either GNU Emacs or XEmacs where appropriate.*

1.1 Context

The XEmacs[1] project started almost twenty years ago after a split from GNU Emacs. This is perhaps the most popular "fork" in the whole free software history. Nowadays, the codebase of XEmacs consists of roughly 400,000 lines of C code [C, 1999] (including the implementation of the Lisp engine) and 200,000 lines of Emacs Lisp. In addition to that, the so-called "Sumo" packages, a collection of widely used third-party libraries distributed separately, amounts to more than 1,700,000 lines of Emacs Lisp code.

Over the years, XEmacs has considerably diverged from GNU Emacs. While the XEmacs development team tries to maintain compatibility for the common functions of the Lisp interface, some features, also accessible at the Lisp level, are exclusive to XEmacs (*extents* and *specifiers* are two common examples). Because of these differences between both editors, writing truly portable code is not easy. In some cases, the existence of compatibility layers makes the task a bit easier. For instance, the implementation of *overlays* (logical parts of text with local

[1] `http://www.xemacs.org`

properties) in XEmacs is in fact designed to wrap around *extents*, our native and more or less equivalent feature.

There is one place, however, in which compatibility with GNU Emacs is neither required nor a problem: the implementation of the editor itself, including its Lisp dialect. After almost twenty years of independent development, it is safe to say that the internals of XEmacs have very little left in common with the original codebase, let alone with the current version of GNU Emacs.

One thing that should immediately strike a newcomer to the internals of XEmacs is the very high level of abstraction of its design. For instance, many editor-specific concepts are available at the Lisp layer: windows, buffers, markers, faces, processes, charsets *etc.*. In XEmacs, every single one of these concepts is implemented as an opaque Lisp type with a well-defined interface to manipulate it. In the current development version, there is 111 such types, 35 of which are visible at the Lisp level (the rest being used only internally).

The other important point here is that although the core of the editor is written in C, there is a lot of infrastructure for data abstraction and sometimes even for object orientation at this level as well. The examples given below should clarify this.

Polymorphism. Many data structures in XEmacs provide a rudimentary form of polymorphism and class-like abstraction. For instance, the `console` type is a data structure containing general data members, but also a type flag indicating which kind of console it is (X11, Gtk, tty *etc.*), and a pointer to a set of type-specific console data. Each console type also comes with a set of type-specific methods (in a structure of function pointers) . This provides a form of polymorphism similar to that of record-based object systems [Cardelli, 1988] in which methods belong to classes.

Accessors. Instead of accessing structure members directly, every data type comes with a set of pre-processor macros that abstract away the underlying implementation. For instance, the macro `CONSOLE_NAME` returns the name of the console, whatever the underlying implementation. This might be considered as more typical of data abstraction in general than object-orientation though.

Dynamic method lookup. There is even an embryonic form of dynamic method lookup which looks very much like Objective-C's informal protocols, or (as of version 2.0 of the language), formal protocols with optional methods [Apple, 2009]. For instance, in order to "mark" a console, without knowing if it even makes sense for that particular console type, one would *try* to call the `mark_console` method like this:

```
MAYBE_CONMETH (console, mark_console, ...);
```

The XEmacs internals are in fact so much object-oriented that the author has raised the idea of rewriting the core in C++ [C++, 1998] directly several times in the past. However, this issue is still controversial.

1.2 Motivation

It is interesting to note that contrary to the internals of XEmacs, there seem to be much less trace of object-oriented design at the Lisp level (whether in the kernel or in the distributed packages), and when there is, it is also much less apparent. Several hypothesis come to mind, although it could be argued that this is only speculation.

- Most of our Lisp interface is still compatible with that of GNU Emacs, and maintaining this compatibility is an important requirement. This situation is completely different from that of the core, which we are completely free to rewrite as we please.
- The need for object orientation in the Lisp layer might be less pressing than in the core. Indeed, many of the fundamental concepts implemented by the editor are grounded in the C layer, and the Lisp level merely provides user-level interfaces for them.
- The Lisp packages, for an important part, are only a collection of smaller, standalone utilities written by many different people in order to fulfill specific needs, and are arguably of a lower general quality than the editor's core. Emacs Lisp contributors are numerous and not necessarily skilled computer scientists, as they are primarily Emacs *users*, trying to extend their editor of choice, and often learning Emacs Lisp on the occasion. Besides, it wouldn't be their job to provide a language-level feature such as a proper object system.
- Finally, it can also be argued that Lisp is so expressive that an object system (whether proper or emulated) is not even necessary for most packages, as similar features can be hacked away in a few lines of code. In other words, it is a well-known fact that good quality code requires more discipline from the programmer, and that the "quick'n dirty" programming paradigm is on the other hand very affordable, and unfortunately widely used.

These remarks make up for the first motivation in favor of a true object system: the author believes that if provided with such a tool, the general quality, extensibility and maintainability of the Lisp code would improve. More specifically:

- Existing C-based features could provide a true object-oriented interface to the user, in coherence with what they are at the C level.
- Lisp-based features would also greatly benefit from a true object-oriented implementation. The author can think of several ones, such as the `custom` interface, the widget code, the support for editing mode and fontification *etc.*

– Finally, the potential gain is also very clear for some already existing third-party packages. The author thinks especially of Gnus[2], a widely used mail and news reader that he also helps maintaining. This package is almost as large as the whole XEmacs Lisp layer, and provides concepts (such as *backends*) that are object-oriented almost by definition.

Once the gain from having a true Emacs Lisp object system asserted, the next question is obviously which one. We are far from pretending that there is only one answer to this question. Emacs Lisp being an independent Lisp dialect, we could even consider designing a brand new object system for it. However, the author has several arguments that would go in favor of CLOS [Bobrow et al., 1988, Keene, 1989], the object system of the Common Lisp language [ANSI, 1994].

– Emacs Lisp is a dialect of Lisp mostly inspired from MacLISP [Moon, 1974, Pitman, 1983] but also by Common Lisp. There are many similarities between Emacs Lisp and Common Lisp, and because of that, a number of Emacs Lisp developers are in fact familiar with both (the author is one of them). Emacs provides a Common Lisp emulation package known as `cl`. A quick survey of the Sumo packages shows that 16% of the distributed files require `cl` to work. In terms of actual lines of code, the ratio amounts to 27%. Given that these numbers don't even count indirect dependencies, this is far from negligible.

– At least in the subjective view of the author, CLOS is one of the most expressive object system available today, and there is freely available code on which to ground the work.

– There is one Emacs Lisp package that already uses a CLOS-like object system (see section 1.3 on the following page). This package is rather large, as it sums up to almost 70,000 lines of code.

– Having CLOS in Emacs Lisp would considerably simplify the porting of existing Common Lisp libraries that could be potentially useful in Emacs. It could also be a way to attract more Common Lisp programmers to Emacs Lisp development.

– CLOS is already very well documented (books, tutorials *etc.*) and we can take advantage of that.

– Last but not least, the author is interested in gaining expertise in the design and implementation of CLOS and its accompanying meta-object protocol [Paepcke, 1993, Kiczales et al., 1991] (MOP for short). Implementing one is a very good way to achieve this goal.

[2] `http://www.gnus.org`

1.3 Alternatives

The author is aware of two alternative object systems for Emacs Lisp.

- The first one is called EOOPS [Houser and Kalter, 1992] (Emacs Object-Oriented Programming System). It implements a class-based, single inheritance, object system with explicit message passing in the vein of Smaltalk-80 [Goldberg and Robson, 1983]. This system dates back to 1992 and the code doesn't seem to have evolved since then. None of the Sumo packages use it and we are not aware of any other Emacs Lisp library using it either.
- The second one is called EIEIO (Enhanced Implementation of Emacs Interpreted Objects). It is part of the CEDET[3] package (Collection of Emacs Development Environment Tools). EIEIO is more interesting to us because our goals are similar: it *is* designed to be a CLOS-like object system. EIEIO provides interesting additional features like debugging support for methods, but apparently, it doesn't aim at being *fully* CLOS-compliant.

The remainder of this paper is as follows. In section 2, we describe the first stage of this project, consisting in a port of Closette to Emacs Lisp. An overview of the differences between Common Lisp and Emacs Lisp is provided, as well as a more detailed description of how the most problematic issues are solved. In section 3 on page 9, we describe how a deeper integration between types and classes is achieved, with respect to what Closette originally offers. Finally, section 4 on page 13 provides an overview of the features available in **CLOX**, and compares the project with EIEIO.

2 Closette in Emacs Lisp

In order to combine the goals of providing CLOS in XEmacs and learning more about its internals at the same time, starting from "Closette" seemed like a good compromise. Closette is an operational subset of CLOS described in "The Art of the Meta-Object Protocol" [Kiczales et al., 1991] (AMOP for short) and for which source code is available. Consequently, Closette constitutes a convenient base on which to ground the work, without starting completely from scratch. The first step of this project was hence to port Closette to Emacs Lisp. This section describes the most interesting aspects of the porting phase.

2.1 Emacs Lisp *vs.* Common Lisp

While both dialects are similar in many ways, there are some important differences that can make the porting somewhat tricky at times.

[3] `http://cedet.sourceforge.net/eieio.shtml`

2.1.1 Fundamental differences

"Fundamental" differences are obvious ones that might require deep changes in the code. The following differences are considered fundamental: Emacs Lisp is dynamically scoped instead of lexically scoped, has no package system, a different condition system, a limited lambda-list syntax, a different and reduced `format`*ing* and printing facility, and also a different set of types.

2.1.2 Subtle differences

"Subtle" differences are less important ones, but which on the other hand might be less obvious to spot. For instance, some functions (like `special-operator-p` *vs.* `special-form-p`) have different names in the two dialects (this particular case is now fixed in XEmacs). Some others like `defconst` *vs.* `defconstant` have similar names but in fact different semantics. Some functions (like `mapcar`) have the same name but behave differently.

Another example is the `function` special operator which returns a functional value in Common Lisp but simply returns its argument unevaluated in Emacs Lisp. In fact, in Emacs Lisp, `function` is just like `quote` except that the byte-compiler may compile an expression quoted with `function`.

The fact that `function` in Emacs Lisp behaves like `quote` might be puzzling for a Common Lisp programmer, but this is because Emacs Lisp accepts a list beginning with `lambda` as a function designator. For instance, the following two lines are equivalent *and* valid for an Emacs Lisp interpreter, whereas the first one would fail in Common Lisp:

```
(funcall  '(lambda (x) x) 1)
(funcall #'(lambda (x) x) 1)
```

2.1.3 Historical differences

In addition to that, Emacs Lisp is still evolving (this appears to be the case in both GNU Emacs and XEmacs) and the changes are not always clearly documented, if at all. For instance, Emacs Lisp keywords were not self-evaluating before 1996, the `#'` syntax exists since XEmacs 19.8 only, characters were not a primitive type until XEmacs 20, and until August 2009, Common Lisp style multiple values were emulated with lists (they are now built-in). **CLOX** is not interested in maintaining backward compatibility with legacy versions of XEmacs or Emacs Lisp, and to be on the safe side, running it typically requires a recent checkout of the 21.5 Mercurial repository[4] (no later than beta 29).

[4] `http://xemacs.org/Develop/hgaccess.html`

2.2 The `cl` package

Emacs provides a Common Lisp emulation package called "`cl`", which is of a tremendous help for porting code from Common Lisp to Emacs Lisp. For the most part, this package provides a number of utility functions or macros that belong to the Common Lisp standard but are not available in raw Emacs Lisp (the almighty `loop` macro is one of them). A number of already existing (but limited) functions are extended to the full Common Lisp power, in which case their names are suffixed with a star (*e.g.* `mapcar*`). `cl` provides `defun*`, `defmacro*` *etc.* to enable the full Common Lisp lambda-list syntax, a version of `typep` and generalized variables via `setf`, `defsetf` *etc.* (`cl`, however, does not support more modern "setf functions").

The remainder of this section provides more details on a couple of interesting porting issues.

2.2.1 Dynamic *vs.* lexical scoping

Perhaps the most important difference between Common Lisp and Emacs Lisp is the scoping policy. Emacs Lisp is dynamically scoped while Common Lisp has lexical scope by default. `cl` provides a construct named `lexical-let` (and its corresponding starred version) that simulates Common Lisp's lexical binding policy via `gensym`'ed global variables. While a brutal replacement of every single `let` construct in Closette would have been simpler, a careful study of the code reveals that this is unnecessary for the most part.

First of all, in the majority of the cases, function arguments or `let` bindings are used only locally. In particular, they are not propagated outside of their binding construct through a lambda expression. Consequently, there is no risk of variable capture and Emacs Lisp's built-in dynamically scoped `let` form is sufficient (it also happens to be more efficient than `lexical-let`).

Secondly, many cases where a true lexical closure is normally used actually occur in so-called "downward funarg" situations. In such situations, the closure is used only during the extent of the bindings it refers to. Listing 1 on the following page gives such an example. The extent of the variable `required-classes` is that of the function. However, the lambda expression that uses it (as a free variable) only exists within this extent. Note that this situation is *not* completely safe, as accidental variable capture could still occur in `remove-if-not`. With a proper naming policy for variables, this risk is considerably reduced, although not completely avoided. In particular, the `cl` package adopts a consistent naming convention (it uses a `cl-` prefix) so that true lexical bindings are unnecessary in practice.

There is a third case in which true lexical bindings can still be avoided, although the situation is an "upward funarg" one: a function that *is* being returned

```
(defun compute-applicable-methods-using-classes (gf required-classes)
  #|...|#
    (remove-if-not #'(lambda (method)
                       (every #'subclassp
                              required-classes
                              (method-specializers method)))
                   (generic-function-methods gf))
  #|...|#)
```

Listing 1: Downward funarg example

and hence might be used outside of the (dynamic) extent of the bindings it refers to. Listing 2 on the next page shows three different versions of the same function.

1. The first one is the original one from Closette. You can see that the returned function has a closure over two variables, which are lexically scoped and have indefinite extent: `methods` and `next-emfun`.

2. The second one follows the logical course of action in Emacs Lisp, using `cl`'s `lexical-let` construct. Recall that function arguments are dynamically bound as well, so we also need to lexically rebind the `methods` variable. Bindings established by `lexical-let` are garbage-collected when the last reference to them disappears, so they indeed get indefinite extent.

3. It turns out, however, that we can still avoid lexical bindings here, by partially evaluating the lambda expression before returning it, as demonstrated in the third version. Remember that in Emacs Lisp, a lambda expression is in fact a self-quoting form, and simply returns a list with the symbol `lambda` in its `car`. Since `methods` and `next-emfun` happen to be constants here, we can pre-evaluate them before returning the lambda expression, hence getting rid of their names which should have been lexically scoped. Finally, note that it is not possible to use `function` or `#'` on a `quasiquote`'ed form, so one might want to call `byte-compile` explicitely on the resulting anonymous function.

Given these particular cases, it turns out that there are only half a dozen places where true lexical bindings are necessary.

2.2.2 Full blown Common Lisp lambda-lists

Because Emacs Lisp is restricted to mandatory, `&optional` and `&rest` arguments, the `cl` package provides replacements for `defun`, `defmacro` *etc.* ***CLOX*** uses these wrappers extensively for its own code, but the question of lambda-lists for generic functions and methods arise. By digging into the internals of `cl`, we are able to provide that at little development cost.

```
(defun compute-primary-emfun (methods)
  (if (null methods)
      nil

      ;; Common Lisp version:
      (let ((next-emfun (compute-primary-emfun (cdr methods))))
        #'(lambda (args)
            (funcall (method-function (car methods)) args next-emfun)))

      ;; Lexically scoped Emacs Lisp version:
      (lexical-let ((methods methods)
                    (next-emfun (compute-primary-emfun (cdr methods))))
        #'(lambda (args)
            (funcall (method-function (car methods)) args next-emfun)))

      ;; Partially evaluated Emacs Lisp version:
      (let ((next-emfun (compute-primary-emfun (cdr methods))))
        `(lambda (args)
           (funcall (method-function ',(car methods)) args ',next-emfun)))
```

Listing 2: Upward funarg example

Internally, `cl` uses a function named `cl-transform-lambda` to both reduce a full blown Common Lisp lambda-list into what Emacs Lisp can understand, and provide the machinery needed for binding the original arguments. Listing 3 on the following page shows an example of lambda-list transformation. Note that the purpose of the second `let` form is to check for the validity of keyword arguments, and disappears if `&allow-other-keys` is provided. In the ***CLOX*** function `compute-method-function`, we take care of wrapping method bodies into a call to `cl-transform-lambda`, hereby providing generic functions with full blown Common Lisp lambda-lists.

Internally (for efficiency reasons), `cl-transform-lambda` uses `memq` to retrieve keyword arguments and hence looks for them at odd-numbered locations as well as even-numbered ones. The drawback of this approach is that a keyword parameter cannot be passed as data to another keyword. Since this does not appear to be much of a problem, we didn't do anything to fix this. This could change in the future, under the condition that the performance of keyword processing in ***CLOX*** does not turn out to be critical.

Also, note that we don't want generic calls to behave differently from normal function calls, so the bindings established by methods remain dynamic.

3 Type/classes integration

Aside from the language differences described in the previous section, the next big challenge to have a working system is to integrate types and classes. This section provides some insight on how this is currently done.

```
;; Original lambda-expression:
(lambda (a &optional (b 'b) &key (key1 'key1))
  BODY)

;; Transformed lambda-expression:
(lambda (a &rest --rest--39249)
  (let* ((b (if --rest--39249 (pop --rest--39249) (quote b)))
         (key1 (car (cdr (or (memq :key1 --rest--39249)
                             (quote (nil key1)))))))
    (let ((--keys--39250 --rest--39249))
      (while --keys--39250
        (cond ((memq (car --keys--39250)
                     (quote (:key1 :allow-other-keys)))
               (setq --keys--39250 (cdr (cdr --keys--39250))))
              ((car (cdr (memq :allow-other-keys --rest--39249)))
               (setq --keys--39250 nil))
              (t
               (error "Keyword argument %s not one of (:key1)"
                      (car --keys--39250))))))
    BODY))
```

Listing 3: Lambda-list transformation example

3.1 Built-in types

As mentioned in the introduction, XEmacs has many opaque Lisp types, some resembling those of Common Lisp (*e.g.* numbers), some very editor-specific (*e.g.* buffers). In XEmacs, there are two basic Lisp types: integers and characters. All other types are implemented at the C level using what is called "lrecords" (Lisp Records). These records include type-specific data and functions (in fact, function pointers to methods for printing, marking objects *etc.*) and are all cataloged in an `lrecord_type` enumeration. It is hence rather easy to keep track of them.

Some of these built-in types, however, are used only internally and are not supposed to be visible at the Lisp layer. Sorting them out is less easy. The current solution for automatic maintenance of the visible built-in types is to scan the `lrecord_type` enumeration and extract those which provide a type predicate function at the Lisp level (the other ones only have predicate *macros* at the C level). Provided that the sources of XEmacs are around, this can be done directly in a running session in less than 30 lines of code.

3.2 Type predicates

3.2.1 `type-of`

Emacs Lisp provides a built-in function `type-of` which works for all built-in types. Because this function is built-in, it doesn't work on ***CLOX*** (meta-)objects. It will typically return `vector` on them, because this is how they are implemented (Closette uses Common Lisp structures, but these are unavailable in Emacs Lisp

so the `cl` package simulates them with vectors). In theory, it is possible to wrap this function and emulate the behavior of Common Lisp, but this has not be done for the following reasons.

- Firstly, we think it is better *not* to hide the true nature of the Lisp objects one manipulates. What would happen, for instance, if a ***CLOX*** object was passed to an external library unware of ***CLOX*** and using `type-of` ?
- Secondly, having a working `type-of` is not required for a proper type/class integration (especially for method dispatch).
- Finally, since ***CLOX*** is bound to be integrated into the C core at some point, this problem is likely to disappear eventually.

3.2.2 `typep`

Having an operational `typep` is more interesting to us, and in fact, the `cl` package already provides it. `cl`'s `typep` is defined such as when the requested type is a symbol, a corresponding predicate function is called. For instance, `(typep obj 'my-type)` would translate to `(my-type-p obj)`.

In order to enable calls such as `(typep obj 'my-class)`, we simply need to construct the appropriate predicate for every defined class. In ***CLOX***, this is done inside `ensure-class`. The predicate checks that the class of `obj` is either `my-class` or one of its sub-classes.

In Common Lisp, `typep` works on class *objects* as well as class *names*. This is a little problematic for us because class objects are implemented as vectors, so `typep` won't work with them. However, we can still make this work in a not so intrusive way by using the `advice` Emacs Lisp library. Amongst other things, this library lets you wrap some code around existing functions (not unlike `:around` methods in CLOS) without the need for actually modifying the original code. ***CLOX*** wraps around the original type checking infrastructure so that if the provided type is in fact a vector, it is assumed to be a class object, and the proper class predicate is used.

3.2.3 Generic functions

Generic functions come with some additional problems of their own. In Common Lisp, once you have defined a generic function named `gf`, the generic function *object* returned by the call to `defgeneric` is the *functional value* itself (a *funcallable* object). In Emacs Lisp, this is problematic for two reasons.

1. Since generic functions are objects, they are implemented as vectors. On the other hand, the associated functional value is the generic function's discriminating function, which is different.

2. Moreover, Emacs Lisp's `function` behaves more or less like `quote`, so it will return something different from `symbol-function`.

In order to compensate for these problems, ***CLOX*** currently does the following.

– A function `find-generic-function*` is defined to look for a generic function (in the global generic function hash table) by name (a symbol), functional value (the discriminating function, either interpreted or byte-compiled), or directly by generic function object (note that this can be very slow).

– Assuming that a generic function is defined like this:

```
(setq mygf (defgeneric gf #|...|#))
(typep mygf 'some-gf-class) ;; already working correctly
```

Class predicates (most importantly for generic function classes) are made to decide whether the given object denotes a generic function in the first place, allowing for the following calls to work properly as well:

```
(typep (symbol-function 'gf) 'some-gf-class)
(typep #'gf 'some-gf-class) ;; #'gf is more or less like 'gf
```

Note that thanks to the advice mechanism described in section 3.2.2 on the previous page, these calls will also work properly when given a generic function class *object* instead of a *name*.

– `find-method` is extended in the same way, so that it accepts generic function objects, discriminating functions or even symbols.

– Finally, ***CLOX*** defines a `function` class. Consequently, in order for `(typep obj 'function)` to work properly, a second advice on the type checking mechanism is defined in order to try the `function` class predicate first, and then fallback to the original `functionp` provided by Emacs Lisp.

Our integration of generic functions into the type system has currently one major drawback: it is impossible as yet to specialize on functions (either generic, standard, or even built-in). The reason is a potential circularity in `class-of`, as described below.

In order to be able to specialize on functions, we need `class-of` to call `find-generic-function*`. However, `find-generic-function*` might need to access a generic function's discriminating function which is done through `slot-value`, which, in turn, calls `class-of`. This problem is likely to remain for as long as ***CLOX*** generic function objects are different from their functional values. In other words, it is likely to persist until the core of ***CLOX*** is moved to the C level.

4 Project Status

In this section, we give an overview of the current status of **CLOX**, and we also position ourselves in relation to Eieio.

4.1 Available Features

As mentioned earlier, stage one of the project consisted in a port of Closette to Emacs Lisp. As such, all features available in Closette are available in **CLOX**. For more information on the exact subset of Clos that Closette implements, see section 1.1 of the Amop. In short, the most important features that are still missing in **CLOX** are class redefinition, non-standard method combinations, `eql` specializers and `:class` wide slots.

On the other hand, several additional features have been added already. The most important ones are listed below.

- **CLOX** understands the `:method` option in calls to `defgeneric`.
- Although their handling is still partial, **CLOX** understands all standard options to `defgeneric` calls and slot definitions, and will trigger an error when the Common Lisp standard requires so (for instance, on multiply defined slots or slot options, invalid options to `defgeneric` *etc.*).
- **CLOX** supports the `slot-unbound` protocol and emulates `unbound-slot-instance` and `cell-error-name`. This is because the condition system in Emacs Lisp differs from that of Common Lisp. In particular, Emacs Lisp works with condition *names* associated with data instead of providing condition *objects* with slots.
- **CLOX** supports a `slot-missing` protocol similar to the `slot-unbound` one. In particular, it provides a `missing-slot` condition which Common Lisp doesn't provide. Common Lisp only provides `unbound-slot`.
- **CLOX** provides a full-blown set of the upmost classes in the standard Common Lisp hierarchy, by adding the classes `class`, `built-in-class`, `function`, `generic-function` and `method`. The other basic classes like `standard-object` already exist in Closette.
- Finally, **CLOX** provides an almost complete type/class integration, which has been described in section 3 on page 9.

Eieio is currently farther away from Clos than **CLOX** already is. Eieio is not built on top of the Mop, doesn't support built-in type/class integration and misses other things like `:around` methods, the `:method` option to `defgeneric` calls, and suffers from several syntactic glitches (for instance, it requires slot

definitions to be provided as lists, even if there is no option to them). EIEIO doesn't handle Common Lisp style lambda-lists properly either.

On the other hand, EIEIO provides some additional functionality like a class browser, automatic generation of `TeXinfo`[5] documentation, and features obviously inspired from other object systems, such as abstract classes, static methods (working on classes instead of their instances) or slot protection *ala* C++. The lack of a proper MOP probably justifies having these last features implemented natively if somebody needs them.

4.2 Testing

Given the subtle differences between Common Lisp and Emacs Lisp (especially with respect to scoping rules), the initial porting phase was expected to be error-prone. Besides, bugs introduced by scoping problems are extremely difficult to track down. This explains why a strong emphasis has been put on correctness from the very beginning of this project. In particular, we consider it very important to do regular, complete and frequent testing. This discipline considerably limits the need for debugging, which is currently not easy for the following reasons.

- **CLOX** is not equipped (yet) for `edebug`, the Emacs interactive debugger, so we can't step into it.
- **CLOX** is not (yet) grounded in the C layer of XEmacs, so we have to use the regular printing facility for displaying (meta-)objects. However, the circular nature of **CLOX** requires that we limit the printer's maximum nesting level, hereby actually removing potentially useful information from its output. We also have experimented situations in which XEmacs itself crashes while attempting to print a **CLOX** object.
- Finally, most of the actual code subject to debugging is cluttered with *gensym*'ed symbols (mostly due to macro expansion from the Common Lisp emulation package) and is in fact very far from the original code, making it almost unreadable. See listing 3 on page 10 for an example.

Apart from limiting the need for debugging, a complete test suite also has the advantage of letting us know exactly where we stand in terms of functionality with respect to what the standard requires. Indeed, tests can fail because of a bug, or because the tested feature is simply not provided yet.

When the issue of testing came up, using an existing test suite was considered preferable to creating a new one, and as a matter of fact, there is a fairly complete one, written by Paul Dietz [Dietz, 2005]. The GNU ANSI Common Lisp test

[5] `http://www.gnu.org/software/texinfo`

suite provides almost 800 tests for the "Objects" section of the Common Lisp standard, but there are also other tests that involve the object system in relation with the rest of Common Lisp (for instance, there are tests on the type/class integration). Currently, we have identified more than 900 tests of relevance for **CLOX**, and we expect to find some more. Also, note that not all of the original tests are applicable to **CLOX**, not because **CLOX** itself doesn't comply with the standard, but because of radical differences between Common Lisp and Emacs Lisp.

The test suite offered by Paul Dietz is written on top of a Common Lisp package for regression testing called "`rt`" [Waters, 1991]. Given the relatively small size of the package (around 400 lines of code), we decided to port it to Emacs Lisp. All porting problems described in section 2 on page 5 consequently apply to `rt` as well. The result of this port is an Emacs Lisp package of the same name, which is available at the author's web site[6]. The test suite itself also needed some porting because it contains some infrastructure written in Common Lisp (and some Common Lisp specific parts), but the result of this work is that we now have the whole 900 tests available for use with **CLOX**.

As of this writing, **CLOX** passes exactly 416 tests, that is, a little more than 50% of the applicable test suite. It is important to mention that the tests that currently fail are all related to features that are not implemented yet. In other words, all the tests that *should* work on the existing feature set actually pass. EIEIO, on the other hand, passes only 115 tests, that is, around 12% of the applicable test suite. As far as we could see, many of the failures are due the lack of type/class integration.

4.3 Performance

As mentioned earlier, we are currently giving priority to correctness over speed and as such, nothing premature has been done about performance issues in **CLOX** (in fact, the performance is expected to be just as bad as that of Closette). Out of curiosity however, we did some rough performance testing in order to see where we are exactly, especially with respect to EIEIO. Figure 1 on the next page presents the timing results of five simple benchmarks (presented on a logarithmic scale). These benchmarks are independent from each other and shouldn't be compared. They are presented in the same figure merely for the sake of conciseness. The five benchmarks are as follows.

1. 1000 calls to `defclass` with two super-classes, each class having one slot.
2. 1000 calls to `defgeneric` followed by 3 method definitions.
3. 5,000 calls to `make-instance` initializing 3 slots by `:initarg`'s.

[6] `http://www.lrde.epita.fr/~didier/software/xemacs.php`

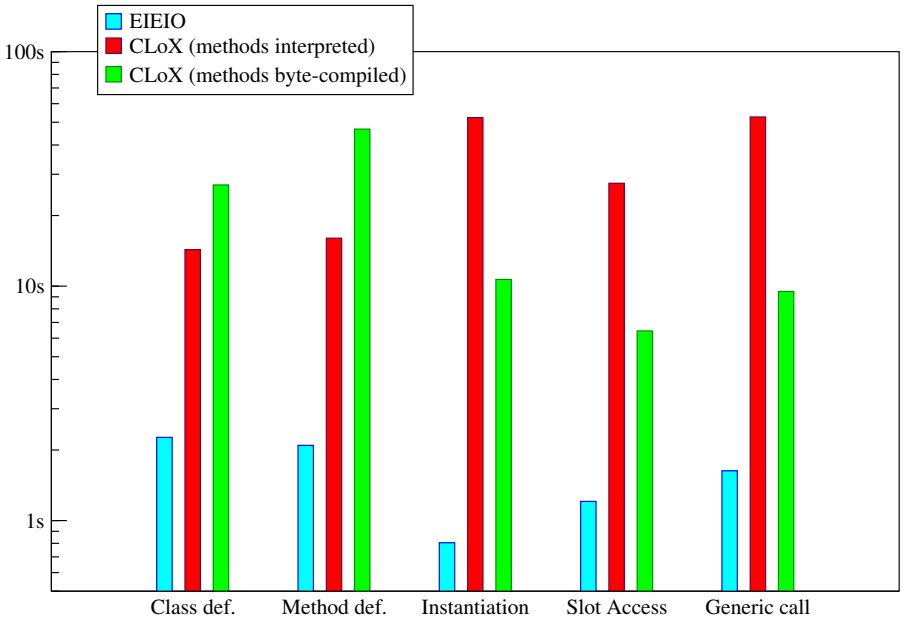


Figure 1: *CLOX* *vs.* EIEIO performance

4. 5,000 calls to 3 slot accessors as defined above.
5. 5,000 calls to a generic function executing 3 methods via 2 calls to `call-next-method`.

These benchmarks have been executed in 3 situations each: once for EIEIO, and twice for ***CLOX***, with method bodies and "upward funargs" either interpreted or byte-compiled.

As expected, EIEIO performs much faster than ***CLOX*** in general. Several specificities of these results can be analyzed as follows.

– Class, generic function and method creation are faster in EIEIO by a factor ranging from 7 to 23. This could be due to the fact that EIEIO doesn't have a MOP, so these operations go through ordinary functions (classes are implemented as vectors).

– When method bodies and other lambda expressions are byte-compiled, ***CLOX*** performs the operations above between 2 and 3 times slower. This is precisely because the results include the time used for byte-compilation.

– On the other hand, the situation is reversed for instantiation, slot access and

generic calls, as they involve executing byte-code instead of interpreting the original Emacs Lisp code. Here, the gain is roughly a factor of 5.

– Finally, we can see that with method bodies byte-compiled (which is the case in EIEIO), instantiation in ***CLOX*** is roughly 10 times slower than in EIEIO, while slot-access and generic calls are about 5 times slower only. Given that EIEIO is already optimized and does not go through a MOP, these results are better than what the author expected.

In order to improve the performance of ***CLOX*** in the future, several paths are already envisioned.

– First, it is possible to implement a caching mechanism and memoize different computation results within the MOP. The AMOP even describes the exact conditions under which a memoized value can be used at some places, for instance in the specification for `compute-discriminating-function` (p. 175).

– Next, there is already abundant litterature on how to improve the efficiency of CLOS (see for example [Kiczales and Rodriguez Jr., 1990]). We can benefit from that experience and also get inspiration from how modern Common Lisp compilers optimize their own implementation.

– Finally, when the core of ***CLOX*** is moved to the C layer of XEmacs, an important immediate speedup is also expected.

5 Conclusion

In this paper, we described the early stages of development of ***CLOX***, an attempt at providing a full CLOS implementation for XEmacs. Details on the porting of Closette to Emacs Lisp have been provided, as well as some insight on type/class integration and how ***CLOX*** compares to EIEIO.

In this project, priority has been given to correctness over speed from the very beginning, which lead us to port `rt` (a Common Lisp library for regression testing) to Emacs Lisp, and also import an important part of Paul Dietz's GNU ANSI Common Lisp test suite. This priority will not change until all the features are implemented properly.

Ultimately, ***CLOX*** will need to be grounded at the C level, at least because this is necessary for a proper integration of generic functions into the evaluator, but also probably for performance reasons.

Once the system is fully operational, the author hopes to convince the other XEmacs maintainers to actually use it in the core, hereby improving the existing code in design, quality, and maintainability. Otherwise, the system will still be useful for third-party package developers willing to use it.

References

[Apple, 2009] Apple (2009). The Objective-C 2.0 programming language. `http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf`.

[Bobrow et al., 1988] Bobrow, D. G., DeMichiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G., and Moon, D. A. (1988). Common lisp object system specification. *ACM SIGPLAN Notices*, 23(SI):1–142.

[C++, 1998] C++ (1998). International Standard: Programming Language – C++. ISO/IEC 14882:1998(E).

[C, 1999] C (1999). International Standard: Programming Language – C. ISO/IEC 9899:1999(E).

[Cardelli, 1988] Cardelli, L. (1988). A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164. A revised version of the paper that appeared in the 1984 Semantics of Data Types Symposium, LNCS 173, pages 51–66.

[Dietz, 2005] Dietz, P. (2005). The Gnu Ansi Common Lisp test suite. In *International Lisp Conference*, Stanford, CA, USA. ALU.

[Goldberg and Robson, 1983] Goldberg, A. and Robson, D. (1983). *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Houser and Kalter, 1992] Houser, C. and Kalter, S. D. (1992). Eoops: an object-oriented programming system for emacs-lisp. *SIGPLAN Lisp Pointers*, V(3):25–33.

[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.

[Kiczales and Rodriguez Jr., 1990] Kiczales, G. J. and Rodriguez Jr., L. H. (1990). Efficient method dispatch in pcl. In *ACM Conference on Lisp and Functional Programming*, pages 99–105. Downloadable version at `http://www2.parc.com/csl/groups/sda/publications/papers/Kiczales-Andreas-PCL/`.

[Moon, 1974] Moon, D. A. (1974). *MacLISP Reference Manual*. MIT, Cambridge, Massachusetts.

[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`.

[Pitman, 1983] Pitman, K. M. (1983). *The Revised MacLISP Manual*. MIT, Cambridge, Massachusetts.

[Ansi, 1994] Ansi (1994). American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999).

[Waters, 1991] Waters, R. C. (1991). Some useful Lisp algorithms: Part 1. Technical Report 91-04, Mitsubishi Electric Research Laboratories.

CLWEB

A literate programming system for Common Lisp

Alexander F. Plotnick
(Brandeis University, USA
plotnick@cs.brandeis.edu)

Abstract: CLWEB is a literate programming system for Common Lisp in the tradition of WEB. It supports program reordering using named sections, code formatting with variable-width typefaces, interactive development, and automatic indexing.
Key Words: literate programming, Common Lisp
Category: D.1, D.3, J.5

1 Introduction

Programming languages are fundamentally aimed at expressing algorithms to two audiences: (1) the computer, via a compiler or interpreter, and (2) the human programmer, whether author, maintainer, or reader. Literate programming is a methodology that gives primacy to the human reader. A literate program reads more like an essay than a program, introducing and using concepts in an order that emphasizes human understanding, regardless of the syntactic constraints of the programming language. It uses natural language to explain the purpose and workings of the code, providing a "mixture of formal and informal methods that nicely reïnforce each other" [9]. It is beautifully typeset, highly readable, and fully indexed. It is still recognizably a computer program, but the primary audience is no longer the machine.

Figure 1 shows an example of a small program written using CLWEB, a new literate programming system for Common Lisp. The program is broken up into numbered *sections*, each of which consists of two parts. First comes the *commentary part*, written in English, followed by the *code part*, written in Lisp. (The programmer does not specify the section numbers or cross-references; they are automatically assigned by the system.)

Every section is either *named* or *unnamed*. A code part that begins with, e.g., '⟨ Section name ⟩ ≡' indicates a definition of the section named 'Section name'. Named sections are essentially parameterless macros; prior to compilation, the system replaces references like '⟨ Section name ⟩' with the code parts of the sections so named. Multiple sections can share a name, which lets you define a section piecemeal: the system appends together the code parts of all the same-named sections and splices them into place when referenced.

1. Buffon's needle. If a needle of unit length is thrown onto a plane ruled with parallel lines spaced one unit apart, the probability of the needle crossing a line is $2/\pi$. Thus, if we simulate a large number of throws, we can obtain an estimate of the value of π.

```
⟨ Return the current estimate of π 1 ⟩ ≡
(return (/ 2.0 (/ hits n)))
```

This code is used in section 2.

2. We'll simulate the needle drop using two random variables: the vertical distance c from the center of the needle to the nearest line below the center, and the sine s of the angle $\theta \in [0, \pi)$ between the needle and the lines.

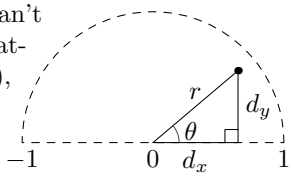
```
(defun estimate-pi (&optional (throws 5000))
  (loop for n below throws
        and c = (random 1.0)
        and s = ⟨ Compute the sine of a random angle 4 ⟩
        count ⟨ Does the needle cross a line? 3 ⟩ into hits
        finally ⟨ Return the current estimate of π 1 ⟩))
```

3. A needle of unit length at angle θ whose center is at $y = c$ will have its ends at $y = c \pm (\sin\theta)/2$. We just have to check if either end crosses either of the lines $y = 1$ or $y = 0$.

```
⟨ Does the needle cross a line? 3 ⟩ ≡
(let ((sin/2 (/ s 2.0)))
  (or (> (+ c sin/2) 1.0)
      (≤ (− c sin/2) 0.0)))
```

This code is used in section 2.

4. Since we're estimating the value of π, we can't just say (*sin* (*random* π))—that would be cheating. Instead, we pick a random point (d_x, d_y), ensure that it lies in the upper half of the unit circle, and use the definition $\sin\theta = d_y/r$.




```
⟨ Compute the sine of a random angle 4 ⟩ ≡
(loop for dx = (1− (random 2.0)) and dy = (random 1.0)
      as r = (sqrt (+ (∗ dx dx) (∗ dy dy)))
      until (< 0 r 1)
      finally (return (/ dy r)))
```

This code is used in section 2.

Figure 1: A sample literate program.

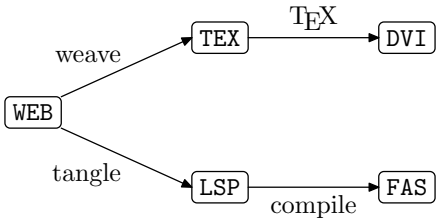


Figure 2: Operation of a literate programming system using Lisp and TEX.

The source file of a literate program, called a *web*, consists primarily of a combination of two languages: a document formatting language for commentary, and a programming language for code. In CLWEB, those languages are plain TEX and Common Lisp, respectively. There's also a third, much smaller language of *control codes* used to delimit sections and their parts, to reference named sections, *&c.* In CLWEB, these all start with '@x', where x is a single character. For example, §2 of the sample program was specified as:

```
@ We'll simulate the needle drop using two random variables:
the vertical distance~$c$ from the center of the needle to
the nearest line below the center, and the sine~$s$ of the
angle~$\theta\in[0,\pi)$ between the needle and the lines.
@l
(defun estimate-pi (&optional (throws 5000))
  (loop for n below throws
        and c = (random 1.0)
        and s = @<Compute the sine of a random angle@>
        count @<Does the needle cross a line?@> into hits
        finally @<Return the current estimate of $\pi$@>))
```

The section begins with the control code '@ ' (space), followed by the commentary. Then comes the code part, introduced with '@l' ('l' for 'Lisp'). (Had this been a named section, the code part would have begun with, e.g., '@<Section name@>=' instead.) In the code, references to named sections are delimited by the control codes '@<' and '@>'.

There are two main operations that may be performed on a web (see Fig. 2). *Weaving* prepares a literate program for reading by a human: it produces a document like the one shown in Fig. 1, containing typeset commentary and code. *Tangling* goes the other way: it strips out all the commentary, expands references to named sections, and outputs a file ready for compilation. These two operations lie at the heart of every literate programming system.

2 Related Work

The first literate programming system was Don Knuth's WEB [8], which used Pascal and TEX as its programming language and document formatting language, respectively. That system was used to implement TEX [10] and METAFONT [11], which are probably the most widely used literate programs. Notable descendents of WEB include Knuth and Levy's CWEB [12], which used the C programming language instead of Pascal, and FWEB [4], which added support for FORTRAN.

All of those systems are (more or less) tied to particular pairs of languages. Many of the newer literate programming systems, such as noweb [16], nuweb [5], and FunnelWeb [21], are language-independent. These systems are usually much simpler than their WEB-like cousins in both use and implementation, and they offer obvious advantages for programmers who frequently switch languages and for projects that employ multiple languages. However, their strength is also a weakness: they cannot, in general, perform the language-specific tasks of automatic code formatting and indexing.[1]

We'll use the term *code formatting* to describe the process of generating nicely typeset output from source code. (This is usually called *pretty printing*, but using that terminology may cause confusion with the Common Lisp pretty printer.) Code formatting can be as simple as reproducing the input source in a monospaced font, or as complex as the transformations performed by, e.g., the 'Fortify' code formatter for Fortress [2], which produces highly mathematical syntax. The first alternative is by far the most common, since it requires minimal effort; anything more sophisticated usually requires a full parser for the programming language.

The other major language-specific task that a literate programming system might perform is automatic indexing of program identifiers. A good index includes separate entries for each type of object named by a given identifier, and indicates whether that object is being defined or just used in a given section. Again, this task generally requires a full parser, but the value of such an index cannot be overestimated, especially for a long program.

For literate programming in Lisp, the systems of choice have generally been language-agnostic tools such as noweb. Systems like AXWEB [22] and LitLisp [13] offer similar features, but happen to be implemented in Lisp. One interesting exception is the Scribble documentation tool for PLT Scheme [6], which uses that language's powerful extension facilities to provide a documentation system that is itself just another PLT Scheme language. Scribble therefore goes even further than the WEB-like systems towards integrating documentation with code, and provides an interesting model to consider for future work on literate

[1] In fact, noweb can do code formatting and indexing through the use of 'filters'—external scripts invoked during weaving. Such filters are necessarily language-specific, though, so the general point still holds.

programming in other Lisp dialects, such as Common Lisp.

There have been many other literate programming systems, but more common are tools that we might call 'semi-literate programming systems'—systems that offer a weaver, but no tangler. Examples include JavaDoc, 'Literate Haskell', Perl's 'Plain Old Documentation' format, and LaTeX's `doc` package. The rationale usually given for the omission of a tangler is that modern programming languages tend to have a less rigid syntax and offer more avenues for abstraction than languages like Pascal and C. This is true, but not the whole story: named sections offer an axis of abstraction that is in many ways orthogonal to the other forms of abstraction provided by modern languages, and the ability to pull out a chunk of code from an arbitrary place in a program and discuss it independently is still valuable. Tangling also allows top-level forms to be reordered in ways that may not be permitted by the underlying language. In Common Lisp, for instance, defining macros such as *defmacro*, *defvar*, and *defclass* (among others) "have compile-time side effects which affect how *subsequent forms* in the same file are compiled" [3, §3.2.3.1.1 (emphasis added)]. Tangling allows users to side-step such effects and present the program in whatever order they desire.

3 CLWEB

`CLWEB` falls squarely in the `WEB` family of literate programming systems. Its syntax, command codes, feature set, and output style are based on those of `WEB` and `CWEB`, so a user familiar with either of those systems should have little difficulty adapting to `CLWEB`.

The implementation is a literate program, written using itself. At the time of this writing, its woven output runs to some 85 pages, including a 4 page index; the tangled source file is approximately 2,500 lines of Common Lisp. It currently runs only under Allegro Common Lisp, Clozure Common Lisp, and recent versions of SBCL ($\geq$ 1.0.31), but ports to several other Common Lisp implementations are planned. A β-quality release is freely available at `http://www.cs.brandeis.edu/~plotnick/clweb/`.

3.1 Differences from `WEB`

The differences between `CLWEB` and `WEB` are mostly ones of omission. For instance, `CLWEB` does not support the 'middle' part of sections, used for macro definitions, because macros are such an integral part of Common Lisp; it seemed best to treat them like any other piece of Lisp code. It does not support change-files, because the standard *diff* and *patch* tools serve the same purpose. It does not support string pools, because the language has native support for strings.

`CLWEB` does not currently support writing specific sections to an arbitrary file (although support for that feature is planned), but it does support a special kind

of section for tests. Test sections are just like ordinary sections—they may be named or unnamed, and have the same two-part structure—but are tangled and woven separately from the main program, so as not to clutter the compiled output or disrupt the narrative flow of the woven document. The weaver generates references from each test section to the regular section it followed, so it's easy to find the code being tested. It's proven quite useful to specify tests alongside the parts of the program being exercised, and to be able to thoroughly describe what is being tested and why.

3.2 Interface

The interface to CLWEB was designed to feel quite a bit more 'Lispy' than the batch-oriented WEB system. For example, in most literate programming systems, the sequence of steps required to run a program is something like:

$$\text{tangle} \rightarrow \text{compile} \rightarrow \text{run}.$$

In CLWEB, there's a single operation, *tangle-file*, that combines the first two steps; it dumps the tangled code to a file using the printer, then invokes the file compiler on that file. Another operation, *load-web*, is analogous to *cl:load*, but operates on a web instead of a Lisp source file; it loops over all the forms in the tangled code and evaluates each one in turn.

Also included with the distribution is a small Emacs Lisp library that provides a major mode for editing CLWEB programs which supports interactive, incremental development with Inferior Lisp mode, ELI, or SLIME. In addition to the usual Lisp evaluation commands, CLWEB mode provides an *eval-section* command that redefines the section at point; in case that section is named, an argument controls whether the code for that section should be replaced or appended to (for piecemeal definitions). Other features include syntax highlighting, commands to move around by sections, and SLIME shortcuts for tangling and weaving the current file.

At the current time, there is no support for helping the Lisp implementation find definitions in the web source as opposed to the tangled file. This would be nice to have, especially for compiler messages, but because there is no portable way to label the provenance of forms in a file, any such future work will necessarily be implementation-specific.

3.3 Reading

Early in the design of CLWEB, I decided to use the Common Lisp reader for input. Two factors influenced this decision. The most important one was the desire to preserve the flexibility of syntax that standard Common Lisp offers. A

secondary goal was to save a bit of effort by not implementing a full parser for all of Common Lisp. As it turned out, neither goal was completely satisfied.

Essentially, the problem is that a system like CLWEB needs to deal not just with Lisp forms as objects, but also with their representations in source code. The Common Lisp reader is very good at dealing with the former, but simply doesn't provide access to the latter. For example, when the Lisp reader is given the characters '`#o177`', it returns the integer 127; the fact that it was originally specified in octal is gone. But the weaver should preserve this kind of information, along with other similar distinctions, like the use of *nil* *vs.* () and (*quote foo*) *vs.* *'foo*. Other pieces of Common Lisp syntax, like #+ and #−, backquote, and #*S*, are important to preserve even during tangling: since these all read in implementation-specific ways, another implementation might not even be able to read the tangled output if we tried to naïvely dump out the objects returned by the reader.

In order to preserve these distinctions and objects that the reader usually discards, CLWEB overrides nearly all of the standard reader macro functions, providing routines that return "marker" objects that represent things like 'rational in radix *r*' or 'comment with text *foo*'. But while the tangler and weaver know how to deal with these objects, user-supplied reader macro functions that expect Lisp forms might not work correctly. This is an open problem for future work: what kind of interface can a system like CLWEB provide that preserves the extensibility of the reader, while overriding most of the reader's functionality?

3.4 Code Formatting

Like the other members of the WEB family, CLWEB formats code using a variable-width typeface. Unlike those systems, however, CLWEB does not attempt to automatically break or indent lines of source code in the woven output.

The decision to omit automatic line breaking was driven by concern for the user: most programmers simply don't like having their lines broken except where *they* break them [17]. CLWEB's weaver therefore respects the line breaks given in the input, and will never break a line of code on its own.

The lack of automatic indentation is purely pragmatic: it turns out to be extremely hard to correctly and automatically indent Common Lisp code. Whereas block-structured languages have simple, fixed indentation rules, Lisp indentation style is highly variable. While it's true that the standard macros and special operators have well-established indentation conventions, user-defined macros might have their own idiosyncratic conventions. Imagine, for instance, a *string-case* macro, or a wrapper around *defclass*; one would presumably like these to be indented like *case* and *defclass*, respectively, but there's no way for a code formatting system to know that without manual annotation. Even with programmer-supplied hints (e.g., the use of *&body* instead of *&rest*), there will always be

macros that require special-purpose indentation rules, the canonical example being the "extended" *loop* form.

CLWEB chooses instead to rely on the author's indentation, and to approximate that in the woven output. If the output were set using a monospaced font, this would be trivial: we would just need to preserve the whitespace used in the input. But with a variable-width typeface, that approach won't work. Consider the following simple form:

```
(first second
       third)
```

Without access to font metrics, the weaver can't determine the width of the first sub-form, and so it can't produce the correct alignment.

Our solution is to record the starting column of each code form, and use that information to construct a sequence of *logical blocks*—groups of forms that share a common left margin. The weaver then prints the logical blocks using TEX's alignment tabs. The result is a variable-width approximation of the original indentation (the TEX source is shown on the right):

```
(first second                    \+(first &second\cr
     third)                      \+       &third)\cr
```

In this way, even the *loop* facility is handled correctly, with no special cases required.

The CLWEB weaver also performs a few other minor code formatting tasks. It sets keywords and lambda-list keywords in an italic typeface; it prints a few symbols using special characters (e.g., it prints the symbols named '`LAMBDA`' and '`PI`' as 'λ' and 'π', respectively); it sets string literals in a monospaced font; and it prints numbers that have a specified radix with that radix as a subscript (e.g., '`#o177`' is rendered as '177_8', as in some math textbooks).

The question of how much formatting is useful or desirable is an open one. For example, one user of CLWEB asked if it would be reasonable to render '`(aref a i)`' as 'a_i'. Fortress does this, but that language has it as an explicit goal to emulate mathematical notation. CLWEB tries to keep Lisp code looking like Lisp, but offers a few minor typographic cues where it seems appropriate. The code formatter is implemented using the Common Lisp pretty printer, so if a user wants fancier output, they're free to add their own pretty printing functions.

3.5 Indexing

Figure 3 shows the index that CLWEB generated for the sample program in Fig. 1. The indexer constructs entries for local and global functions, macros, and symbol macros; special variables; constants; classes and condition classes; generic functions; methods (with qualifiers); and *setf* functions and methods. It does not

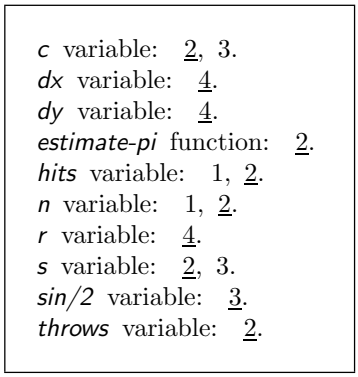


Figure 3: The index for the program shown in Fig. 1. All references are to section numbers; an underlined locator indicates definition in the given section.

usually index lexical variables, on the presumption that they would needlessly bulk up the index, but will do so if a certain global flag is set to true, as it was for the example above.

The usual way of generating an index or cross-reference database for Lisp is to use a *code walker*. This is a tool that knows how each of the 25 special operators of Common Lisp affects the lexical environment of its sub-forms. Given a form and a lexical environment object,[2] a code walker macroexpands until it reaches a special form, then recursively walks each sub-form with an appropriately augmented lexical environment object. The walker generally runs some user-supplied code at each stage of the walk, passing the form being walked and some sort of context, including the current lexical environment.

The basic problem with indexing a web is that the code parts aren't quite Common Lisp: they're fragments of Common Lisp that might contain named section references. You can't walk these with a code walker, because the walker needs to perform macroexpansion, which could break in the presence of named section references. The naïve solution is to tangle the web first, then walk the tangled code. This would be fine from the point of view of the walker, but would defeat the entire purpose of walking the forms in the first place: remember, the idea is to generate a mapping of objects named by symbols or function names to the sections in which references to those objects occur—that's what an index is. But if we tangle before walking, we'd lose the provenance of those occurrences; we'd know that some object was being referenced, but we wouldn't know *where*.

[2] Unfortunately, although Common Lisp mandates the existence of environment objects, it does not supply a standard way of either examining or augmenting them. An API for doing so was proposed in CLtL-2 [19], but was ultimately rejected by the X3J13 committee. Nevertheless, that API is relatively widely supported, and so it is used for CLWEB's code walker. It's the one non-standard dependency in CLWEB.

The solution used in CLWEB is slightly tricky, and more than a little smarmy. Before we walk, we do a special sort of tangling that not only expands named section references, but also replaces every symbol in each section with what we call a *referring symbol*. This is an uninterned symbol whose value cell contains the symbol it replaced, and which has a property set on its plist that contains the section in which the original symbol occurred. Then, as we do the code walk, we use these referring symbols to build the index, and swap them back out for their referents to preserve the semantics of the original code for the next stage of the walk.

This trick does have one flaw, which is that macros that depend on symbol identity may not be expanded correctly during the walk, because they'll see the referring symbols instead of whatever they might be expecting. But since the walk is only done to build the index, unless such a macro signaled an error for some reason, the worst that could happen is that the index might not be as complete as it should be. In practice, this has not been a problem.

CLWEB provides its own code walker, as none of the freely available ones were found to be suitable. The design of the walker comes from Richard Waters's *macroexpand-all* [20], but it's written in a more modern style, making heavy use of CLOS; the indexer proper is just a subclass of the walker with some specialized methods, which makes extending it relatively straightforward.

4 Conclusion

Having described literate programming in general and CLWEB in particular, we should perhaps pause for a moment and ask the question, "Do we really need literate programming in Common Lisp?" As Lisp programmers, we are used to working with a highly expressive language that already has many avenues for syntactic and semantic abstraction. Do we need more?

I think that we do. As elegant and expressive as Lisp is, a complex program still requires a guide. Think, for example, of the first part of *The Art of the Metaobject Protocol* [7]. This contains a nearly complete implementation of a small, CLOS-like language, thoroughly and elegantly described. With a tiny amount of extra work, that could have been recast as a literate program, eliminating the need for a separately-maintained implementation. Indeed, many of the best textbooks (e.g., [1, 14, 15]) present programs in exactly the style of a well-written literate program: broken up into small, manageable pieces that are completely documented. We don't need to struggle to understand these programs, since our comprehension was the principle guiding the presentation.

Literate programming is, I believe, uniquely suited to creating complex programs that exemplify such excellence of style and documentation. I hope that CLWEB will be a useful tool for those who seek to produce such programs.

References

1. Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, second edition, 1996.
2. Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steel, Jr., and Sam Tobin-Hochstadt. The Fortress language specification, March 2008.
3. American National Standards Institute. Committee X3, Information Processing Systems and Computer and Business Equipment Manufacturers Association. *Draft proposed American national standard programming language Common LISP. X3.226:199x. Draft 14.10, X3J13/93-102*. Global Engineering Documents, Washington, DC, USA, January 1994.
4. Adrian Avenarius and Siegfried Oppermann. `FWEB`: A literate programming system for Fortran 8X. *ACM SIGPLAN Notices*, 25(1):52–58, January 1990.
5. Preston Briggs. Nuweb, A simple literate programming tool. `cs.rice.edu:/public/preston`, Rice University, Houston, TX, USA, 1993.
6. Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: closing the book on ad hoc documentation tools. *SIGPLAN Notices*, 44:109–120, August 2009.
7. Gregor Kiczales and Jim des Rivieres. *The art of the metaobject protocol*. MIT Press, Cambridge, MA, USA, 1991.
8. Donald E. Knuth. The `WEB` system of structured documentation. Stanford Computer Science Report CS980, Stanford University, Stanford, CA, September 1983.
9. Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.
10. Donald E. Knuth. *TEX: The Program*, volume B of *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA, 1986.
11. Donald E. Knuth. METAFONT*: The Program*, volume D of *Computers &Typesetting*. Addison-Wesley, Reading, MA, USA, 1986.
12. Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation, Version 3.0*. Addison-Wesley, Reading, MA, USA, 1993.
13. Drew McDermott. Litlisp, a literate-programming system based on 'txtlisp', 2006.
14. Peter Norvig. *Paradigms of artificial intelligence programming: case studies in Common Lisp*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 2004.
15. Christian Queinnec. *LISP in small pieces*. Cambridge University Press, Cambridge, UK, 1996.
16. Norman Ramsey. Literate programming tools need not be complex. Technical report CS-TR-351-91, Princeton University, Dept. of Computer Science, Princeton, NJ, USA, October 1991.
17. Norman Ramsey and Carla Marceau. Literate programming on a team project. Technical report CS-TR-302-91, Princeton University, Dept. of Computer Science, Princeton, NJ, USA, February 1991. Published in [18].
18. Norman Ramsey and Carla Marceau. Literate programming on a team project. *Software—Practice and Experience*, 21(7):677–683, July 1991.
19. Guy L. Steele, Jr. *COMMON LISP: the language*. Digital Press, 12 Crosby Drive, Bedford, MA 01730, USA, second edition, 1990. With contributions by Scott E. Fahlman and others, and with contributions to the second edition by Daniel G. Bobrow and others.
20. Richard C. Waters. Macroexpand-all: an example of a simple lisp code walker. *SIGPLAN Lisp Pointers*, VI(1):25–32, 1993.
21. Ross Williams. FunnelWeb user's manual. `ftp.adelaide.edu.au` in `/pub/compression` and `/pub/funnelweb`, University of Adelaide, Adelaide, South Australia, Australia, 1992.
22. Stephen Wilson. `AXWEB` – a literate programming tool implemented in Common Lisp, July 2007.

Colophon

The papers collected here were formatted according to the style guidelines for the Journal of Universal Computer Science (those using LaTeX using the `jucs2e.sty` style file). These proceedings were typeset using XeLaTeX, including the typeset papers using the pdfpages package. The typefaces used for this collection are the Linux Libertine and Linux Biolinum typefaces from the Libertine Open Fonts Project, and the `Computer Modern Teletype` typeface originally designed by Donald Knuth. The front cover image is a photograph of the *Padrão dos Descobrimentos* (Monument to the Portugese Discoveries) by Georges Jansoone, modified by Edgar Gonçalves, and used here under the terms of the Creative Commons Attribution 2.5 Generic copyright licence.