Proceedings of the First CLOS Users and Implementors Workshop

October 3rd and 4th, 1988
Palo Alto, California

AGENDA

CLOS Workshop
Palo Alto Research Center
Xerox Corporation
October 3rd and 4th, 1988

October 3rd

8:30-9:00 - Registration and Breakfast

Session 1: 9:00 - 10:30

Workshop Goals

Daniel G. Bobrow, Xerox PARC

Inside PCL

Gregor Kiczales, Xerox PARC

Break for Refreshments 10:30 - 11:00

Session 2: 11:00 - 12:30

CLOS and Persistent Storage

Larry Rowe, UC Berkeley
Steve Ford, Texas Instruments
Jim Bennet, Coherent Thought Inc.

Lunch on the Patio 12:30 - 2:00

(vegetarian meals available)

Session 3: 2:00 - 3:30

Environments and Tools
(Live Demonstration Session)

Stan Lanning, Xerox PARC
Ken Anderson and Rick Shapiro, BBN
Jim Kempf, SUN Microsystems

Break for Refreshments 3:30 - 4:00

Session 4: 4:00 - 5:30

Future Implementaton Techniques

Daniel G. Bobrow, Xerox PARC
Masayuki Ida, Aoyama Gakuin University
Patrick Dussud, Lucid Inc.

Banquet Dinner at Chef Chu's

At the corner of San Antonio and El Camino (see map)

(vegetarian meals available)

Open Bar from 7:00 - 7:30
Dinner at 7:30

AGENDA

CLOS Workshop
Palo Alto Research Center
Xerox Corporation
October 3rd and 4th, 1988

October 4rd

8:30-9:00 - Breakfast

Session 5: 9:00 - 10:30

Applications

Harley Davis and Sanjay-Mittal.
Xerox PARC

David Wallace, HP Labs

Angela Sutton, University of Aberdeen and Schlumberger Palo Alto Research

John Davidson, MITRE Washington

Session 7: 2:00 - 3:30

Window Systems and User-Interface Toolkits
(Taped Demonstration Session)

Hans Muller, SUN Microsystems

Robert C. Pettengil, MCC

John Dye, Advanced Decision Systems

Break for Refreshments 10:30 - 11:00

Break for Refreshments 3:30 - 4:00

Session 6: 11:00 - 12:30

Metaobject Status and Applications

Daniel G. Bobrow and Gregor Kiczales, Xerox PARC

Angela Dappert-Farquhar, Siemens

Pierre Cointe, Rank Xerox and L.I.T.P.

Andreas Paepke, HP Labs

Session 8: 4:00 - 5:30

Plans for the Future

Lunch on the Patio 12:30 - 2:00

(vegetarian meals available)

EXPERIENCE WITH OBJECT AND METAOBJECT PROGRAMMING IN PCL
OR
WHAT DO YOU MEAN WHEN YOU SAY "METAOOPS"?

Kenneth R. Anderson
Michael Thome
Richard Shapiro

BN Laboratories Division
BBN Systems and Technologies Corporation
70 Fawcett St., Cambridge MA 02238

BBN has been working with Portable Common Loops, PCL, the prototype of the Common LISP Object System, CLOS, for about two years. Interest and use of PCL at both the object and metaobject levels have increased steadily. We currently have at least seven major projects using PCL. Interesting applications include:

- o KREME - Knowledge Representation, Editing and Modeling Environment
- o PARCL - Extentions to PCL for Parallel and Distributed processing
- o AIIS - A Tax return classification system for the IRS

We have also build several system level tools including:

- o CONDITION - An error handling system.
- o FLOID - A Flavor impersonator
- o A class browser and class lattice grapher.
- o Uncommon Windows - Fully object oriented Common Windows

PCL and CLOS have several unique features that have proven to be valuable in developing these systems:

- o Multimethods - Methods discriminate on multiple arguments (unSELFishness)
- o Discrimination on nonCLOS objects
- o Extendability through its Metaobject Protocol.

These features are likely to be as important to the future of objectkind as, say, inheritance is considered today.

In other object oriented languages, method dispatching depends on only the first argument. This makes the first object artificially superior to the others and makes relationships between objects harder to express. CLOS' multimethods eliminates this problem and encourages programmers to be less object oriented and more behavior or protocol oriented. This better balances the dual nature of object and behavior.

CLOS methods can discriminate on nonCLOS objects provided by the underlying LISP implementation. This allows CLOS methods to be better integrated with LISP, and lets CLOS programmers take advantage of the improved performance of more primitive LISP objects. It also allows to better coexist with foreign object oriented systems such as KEE and Flavors.

The Metaobject Protocol has been used to provide useful extension to PCL, as well as exploring alternative object oriented paradigms. Examples of metalevel extensions include:

- o Instance and slot locking.
- o Active slots.
- o Classes that record their instances.
- o Allocating instances from resources.
- o Delegation.
- o Method discrimination on LISP structures.

The Metaobject Protocol allows new worlds object oriented programming to be explored. New programming issues have already begun to emerge from this exploration, including:

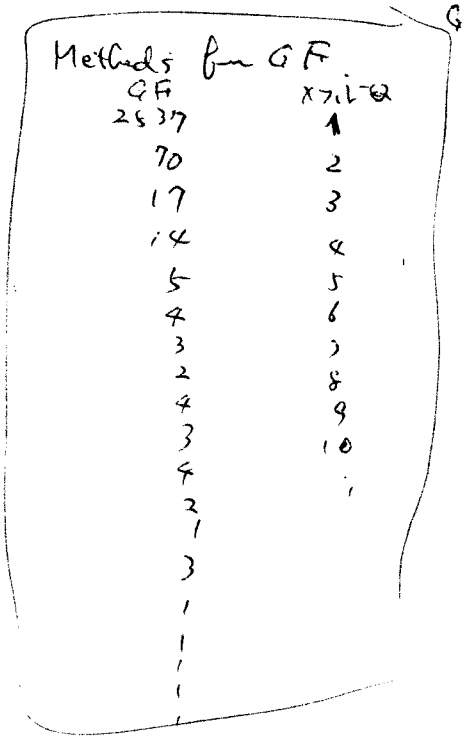
- o Metafairness - Should the metaobject protocol be biased toward any particular object oriented feature or paradigm?

- o Metabrotherhood - To what extent can multiple paradigms coexist in the same programming environment?

- o Metaencapsulation - Does a user need to know what paradigm an object comes from in order to use it or inherit from it.

The ultimate test of the Metaobject Protocol will be if CLOS can be used to write the language that will replace it.

See P105



An implementation of CLOS

Giuseppe Attardi
Maria Rosaria Boscotrecase

I CLOS workshop

Palo Alto, 3-4 october 1988

Abstract

We describe an approach to an efficient CLOS implementation on the basis of a simple semantic model. The kernel of CLOS is implemented in C language to achieve efficiency. A full integration of classes and structure is still provided.

1. Introduction

We describe an implementation of CLOS developed for *DELPHI COMMON LISP*, which is an extension of *KYOTO COMMON LISP*. Being able to modify the underlying CL implementation, we decided to do it whenever we found it convenient. In the following we first define the underlying model and then introduce the implementation issues. In the end we discuss some topics about CLOS specification.

2. The model architecture

The three CLOS primitive metaclasses, *built-in-class*, *structure-class* and *standard-class*, are not all instances of *standard-class* as required in the CLOS specification. We don't feel that *built-in-class* and *structure-class* must be instances of *standard-class*, which should only be the default metaclass of classes created by **defclass**. For example if *built-in-class* is a standard-class, it could be possible to invoke some methods don't make sense for it as **default-initargs**. This is why we chose to implement these three metaclasses as instances of *Class*, which is the primitive metaclass. *Class* is not just an abstract class, but is a real class built as the minimum common structure of any class [ObjVlisp]. For the same reason *built-in-class* and *structure-class* don't inherit from *standard-object*, which describes the default behaviour of standard-classes.

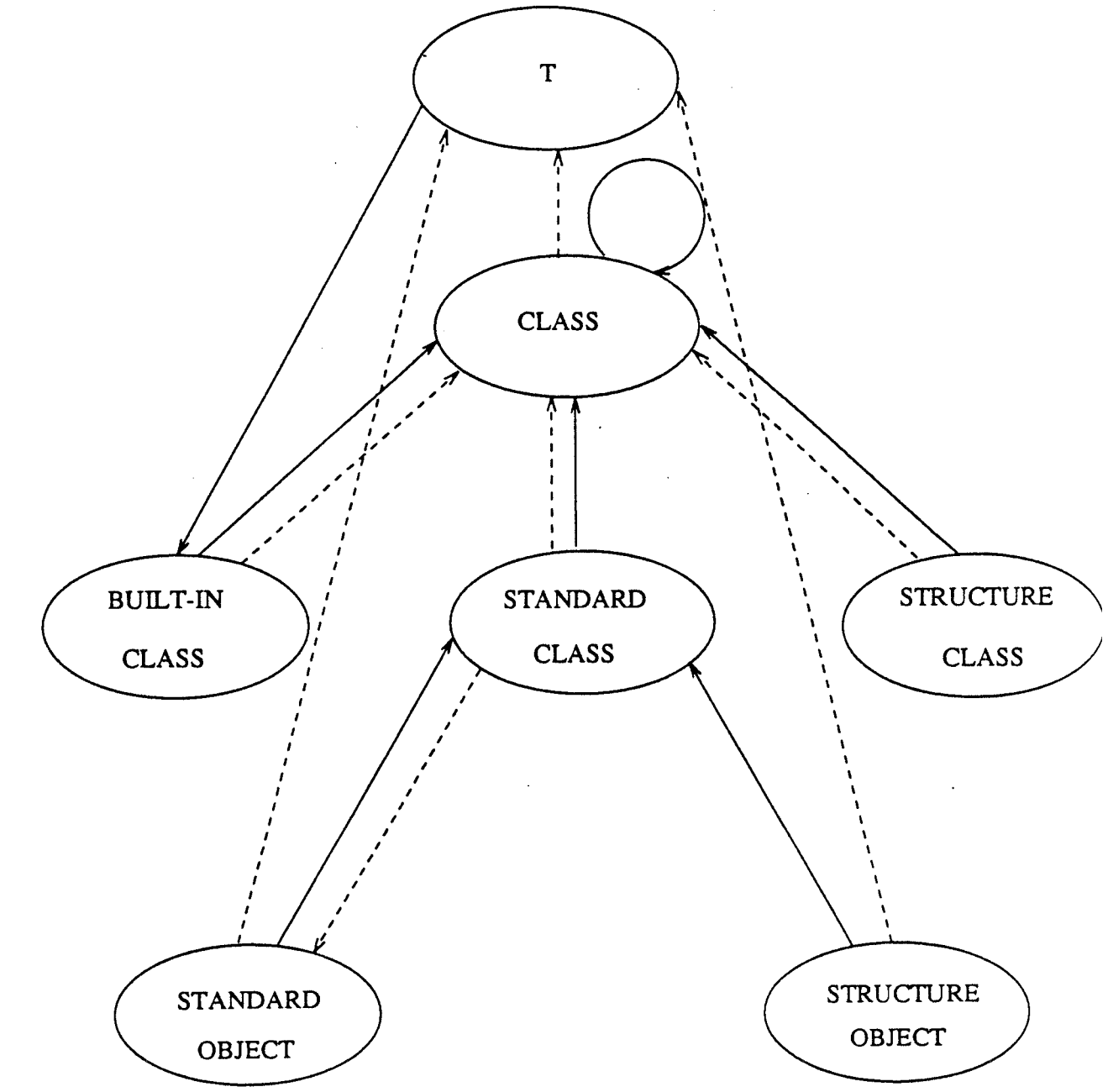
In figure 1 there is the hierarchy among the classes in our kernel.

3. The implementation of an instance

In order to gain efficiency, we introduced a new Lisp data type, named *closobject*, implementing a CLOS instance. This is a C structure, very similar to the one implementing CL structure, containing a pointer to the class of the instance, a pointer to a contigous area storing the slots of the instance and the length of this area.

This approach allows us to easily manipulate CLOS objects inside the kernel of Lisp, for example to implement generic functions. It also supports the redefinition of classes, since it is possible to replace the area containing slots without changing the identity of the object.

FIGURE 1


T
CLASS
BUILT-IN CLASS
STANDARD CLASS
STRUCTURE CLASS
STANDARD OBJECT
STRUCTURE OBJECT
INSTANTIATION
INHERITANCE

4. Generic functions

We introduced a new Lisp type, named *gfun*, that represents generic functions as Lisp objects and is evaluated as a form. This is a C structure containing the generic function name, the generic function instance and some information useful for cashing methods.

When a generic function is invoked, the code to execute depending on the arguments in the call must be determined. This code is called the *effective method* for those arguments. It would be very inefficient if the effective method would be computed each time the generic function is called. In order to avoid this, our implementation uses a caching tecnique. The first time a method on a specific set of arguments is invoked, the effective method is computed and stored in a cache table. The table is accessed by a hash key computed from the parameter specializer names. When a new method is introduced for a generic function the whole cache of the generic function is cleared, so that the next time the function is called on specific parameters, the effective method is recomputed, possibly incorporating the new method because of method combination.

5. The implementation of defstruct

In order to integrate the CL structures in the class graph we have modified the CL **defstruct** macro to define a class for each structure type. This class describes the same slots specified by the **defstruct** option and it is an instance of the class named *structure-class*. The instances of the structure are created as instances of the class associated to the structure and the low level primitives managing structure instances work on these instances. So we have a unique internal representation for instances and structures that contains all the information instead of having two different data types, one for objects and one for structures. In particular one can apply structure accessor functions to objects which inherit from a structure class.

The **defstruct-defclass** integration is essential for supporting CLUE, where **contact** class inherits from **xlib:window**, which is a structure dealt by the CLX package.

Two issues are still open in the field of the integration of classes and structures. The first one concerns inheritance, because a some checking must be done to avoid a class could inherit from more than one structure class. The second one involves the print function of a structure, which has three arguments (object stream and depth) while **print-object** has just the first two.

6. Some suggestions about CLOS

We present a few remarks on the CLOS specification.

The model architecture presented here is slightly different from the one described in Chapter 3 of the CLOS specification, which has different relationship among the primitive metaclasses. We propose to discuss it in order to see which could be the minimal and cleanest model.

CLOS introduces the concept of prototype instance. A prototype instance is an instance of a class and can be a completely blank instance as created by allocate-instance. The only requirements for this instance are that it will respond properly to class-of and it will allow the right selection of the method if passed as argument in generic function call. For example the macro **defclass** expands to a call to the generic function **add-named-class** passing it a prototype instance of the class specified by the :metaclass option in the defclass form. Here the use of the prototype instance allows to invoke methods passing them as argument the class to define before having the class as object. We feel that the introduction of the concept of a prototype instance is an unnecessary complication. It also invites confusion between the new class you are creating and the metaclass which is instantiated to make the prototype instance.We suggest instead that the concept of metaclass is sufficient to implement the desired behaviour simply by invoking *add-named-class* with the metaclass as an argument. Sometimes this approach could require the introduction of new metaclasses just for the purpose of defining an *add-named-class* method, but we don't think this is a significant overhead. For example we had to introduce the classes *standard-metaclass* and *structure-metaclass* just to provide the *add-named-class* method used by their instances *standard-class* and *structure-class* respectively. An alternative solution could also be to define the *add-named-class* method with (**eql** *object)* as specializer for the first argument.

Another topic is about the initialization arguments used during the creation of an object; we suggest that the **:default-initargs** option to **defclass** should not be provided. In fact it introduces an unnecessary complication in the CLOS specification and implementation. It is still redundant because the same behaviour can be obtained giving initforms to keywords in the **initialize-instance** method. Since the order of evaluation of defaults value forms is undefined, its behaviour is unpredictable. Moreover its use is difficult to understand for a user.

We feel that the CLOS specification should not address any issue that is strictly implementation dependent. For example, CLOS forsees a mechanism for optimizing calls to **slot-value** at compile-time. We think that each implementation should be free to provide its own optimization without being bounded by a predefined protocol. Moreover, CLOS specification describes the **shared-initialize** generic function, which fills the slots of an instance through the use of initialization arguments and *:initform* forms. **shared-initialize** is called by the system-supplied methods to initialize a new instance and to reinitialize an old one. Providing a generic function like it means including implementation details in the specification of a standard.

In the end we suggest minimizing the number of concepts introduced by the language. This goal can be obtained without losing the generality and the flexibility of the resulting system.

We have experimented the extendibility of our system implementing the main features of KRS, a concept based system for the implementation of different knowledge representation schemas, in CLOS.

Ryuichi Aoki
System Software Development Department
System Technology Center
Document System Unit
Fuji Xerox Co., Ltd.

We have been developing an intelligent system on PCL. And I am just taking charge of the graphic interface tool. An overview of the graphic interface tool is following. And I have some opinions about CLOS programming environments, extensions, etc. But this paper does not include them.

From the point of application programmer's view, especially from intelligent system builder's eyes, it is important to describe the world that he deals with. Object oriented programming environment, CLOS provide us class-instance methodology for framing simulation models with object system. Then, not only defining of classes and methods, but also organizing objects should be important.

Our graphic interface tool on CLOS intends to support both application builders who organize the object models, and end users who interacts the models. And there are three kinds of view to support for them.

1) View of browsing networks of objects in the model

The relation of objects are visible and editable on the browser using two graphic items, 'link' and 'node'. The program interface to specify the relation is prepared. A developer can define the browser for an arbitrary relation by the program interface. And beforehand the graphics interface tool provides two convenient browsers:

Model browser makes the part-of relations of objects visible and editable. Each node corresponds to a CLOS object or LISP object. A developer is able to organize objects by the browser.
Class browser makes the kind-of relations of classes visible. Each node corresponds to a class. A developer is able to define/redefine classes and methods by the browser.

2) View of focusing an object

Some objects displayed on Graphic interface windows are editable in their proper windows. Specified slots of objects are editable in their proper windows.

3) View of exhibiting variety aspects of object model

Actual object models must be organized complicatedly, to be impossible to display at once. Object panel is the window to see and operate the status of the objects. Things those are displayed as the graphic items on the object panel are programmers' specified. Some graphic items are connected with certain objects of the actual object model, and others are not.

Our user interface tool defines more than twenty classes for visible items. They are categorized into three types, they are :

1) Items which can be connected with an object

These items can describe the status of the connected object by pictorial representation, for example, texture change, bitmap, text, blinking, and so on.

Also value of the connected object can be handled through text editing frame, gauge frame, toggle switch, etc.

2) Items which can activate procedures

Functions, methods can be activated with arguments by clicking on an item. This intends to support procedural manipulation on the object model.

3) Simple graphical image

This tool provides flexible functions on uniform operations. For example, certain object which is connected a graphic item can be copied from/to a panel and a browser with the same operations.
This tool is framed on the CLOS object system dominating Interlisp-D window/graphic system. About forty classes are defined for the tool.

Configuration and Version Management for CLOS-based Products

James Bennett, Chris Clifford, John Dawes
Coherent Thought Inc.
3350 West Bayshore Road, Suite 205
Palo Alto, CA 94303

For discussion at the CLOS Workshop, October 3-4, 1988. On-line queries about these issues may be directed to jbennett@coherent.com.

1 Introduction

Coherent Thought develops problem-specific expert system tools and knowledge base development environments for solving commercial diagnosis, configuration management, and financial risk-assessment problems. Like most expert system tools, the basic tools are independent of application-specific knowledge, however, unlike those tools, our systems provide a basic problem-solving approach and vocabulary for the problem and the system. Exploiting the given structure of the problem-solving approach permits knowledge base development, maintenance, and explanation facilities to be specialized to the particular problem and application in an effective manner.

Each of our products is an extensible "template", built to be customized first to particular application market areas and then, via customer knowledge base development, to specific applications (e.g., diagnosis of electro-mechanical devices, then computer networks or automobiles, etc.). We expect to license our products to professional software application builders (systems integrators, hardware vendors, etc.) as a set of modules that they can embed in their existing or new software. We expect the knowledge bases for these delivery systems to be developed and maintained on workstations using sophisticated knowledge-based knowledge acquisition tools and that the actual systems will be delivered in different environments, including MVS/CICS on IBM mainframes and various UNIX- and OS/2-based workstations.

2 Configuring CLOS-based Systems for Product Release

We construct our product templates using Common Lisp and CLOS. Each template itself is large and complex, and is developed by a multi-person team. The combination of Common Lisp and CLOS provide an extremely flexible software development environment for the construction of our template products. Following the OOPS methodology, we expect to make substantial re-use of portions of our code between templates and to isolate operating environment-specific aspects of the templates, capitalizing on the extensive object-oriented programming metaphor that CLOS

provides. We expect the use of these techniques will permit us to rapidly extend and enhance our product offerings as our customers' requirements change.

The production, documentation, and maintenance of software products, however, requires proper configuration, release control, and tracking of the component elements of the product. Under most object-oriented programming schemes, including CLOS, the number and complexity of potential software configurations can increase quite rapidly, potentially multiplying the burden of tracking and maintaining the state of each product release. In particular, there is a tendency to proliferate a large number of partially-functional components from which final modules might be constructed ("mixin madness"). A tradeoff exists, then, between exploiting the flexibility the OOPS development tool provides for designing the software and managing the product maintenance, testing, and documentation requirements.

To address this tradeoff and create delivery versions of our products, we expect to combine standard release control techniques and systems with OOPS-specific programming disciplines and proprietary techniques for creating and compiling portable delivery versions of the tools in the C and 370-assembly languages. The specific disciplines and tools will, of course, be developed, adapted, and honed on the basis of our particular experience and market requirements. Many of the requirements and principles, based on standard software engineering methodologies, can be anticipated, however, and existing tools can be investigated. We look forward to the CLOS workshop providing a forum to discuss these issues, both in principle and based on practice.

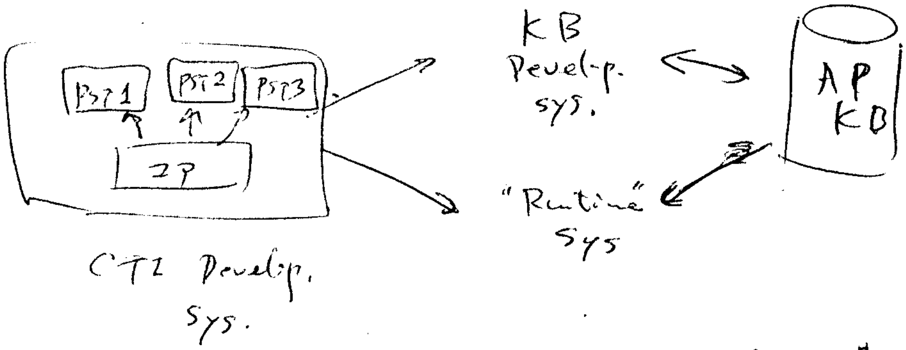
Using a prototype of one of our product templates, we have begun one set of experiments with a configuration discipline and CLOS/LISP-based support tool set. The prototype problem-specific template was developed in CLOS in a relatively unrestricted fashion as a specialization of a shared "core" of basic reasoning and knowledge-base maintenance facilities. The prototype was reviewed and a set of provisional "module" boundaries were identified that correspond roughly to expected released product elements for the purposes of documentation production, inter-facility testing, and update maintenance.

We also chose these boundaries looking to restrict the kinds of inter-module reference and specialization, which in turn would allow us to reduce or eliminate portions of the full CLOS facility in the delivery environment. Based on those initial observations, we have developed a process of converting development CLOS systems into delivery CLOS modules according to the product module declarations. We have also developed an implementation of a CLOS subset to support modules based on these assumptions which is significantly smaller than the current CLOS implementation and which requires significantly reduced overhead to support the final module functionality. This will be especially important in delivery systems where no further development will occur but where efficient use of memory resources are at a premium and execution speed is critical, as is the case in our target transaction processing market. We will discuss some of our initial experimental results at the workshop.

Although our initial experiments are promising, they deal only with the early stages of product configuration, version maintenace and support requirements. We look forward to talking with other system developers at the CLOS workshop on these extended "software lifecycle" problems and their solutions for OOPS- and CLOS-based products. In particular, we would like to discuss:

- Coding styles and development disciplines that aid conversion of prototype and development code to delivery versions of systems
- Ideas and principles for documenting and advertising functionality for OOPS-based systems.
- Experience with version maintenace and configuration of large-scale OOPS-based systems in general.

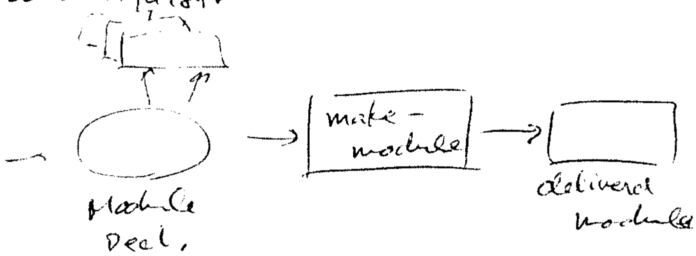
- Experience with existing LISP-based software configuration and tracking systems.
- Experience with existing, non-LISP version maintenance tools (e.g., SCCS, RCS, Amplify, etc.), especially as applied to OOPS-based and LISP-based systems.



Delivered to UNIXベース 及び
MVS/CICSベース
Mixture of CLOS/CL & C.

Config. & Version Control Tradeoffs
・開発は柔軟性がいる
・配布の保守とリリースにはコントロールがいる

CLOS モジュール Experiment
・クラス, GF, スロットを export
・モジュール内のスロットを隠す [illegible]
・基本メソッドのみ export
・モジュール内の 'call-next-method' を [illegible]
・モジュールの依存性についての宣言を示す.



モジュールについての [illegible]
CLtL のモジュールの [illegible]

Evaluation Observation
・Delivered Module は CLOS ランタイムサポートを含まず.
・CLOS サポート 20% 増加 of 標準リリース
・クラス & G.F. [illegible]
・Load time を [illegible]

The Importance of Being Meta

Daniel G. Bobrow & Gregor Kiczales
System Sciences Laboratory
Xerox PARC

Position paper for first CLOS Workshop
October 3rd and 4th 1988

One of the most important capabilities Lisp provides is that when writing a Lisp program it is possible to build a specialized language tailored to the problem at hand. This language can be an extension of the underlying Lisp, and it can be embedded into the underlying Lisp. Programs can be written which use both the traditional Lisp and the specialized language. The resulting code is more perspicuous and is often more efficient as well. An excellent description of the importance of this capability was given by Sussman at the 1988 Lisp Conference [1].

The two major traditional Lisp-Object-Oriented-Languages (Flavors [2][3] and Loops [4]) have not provided this capability. The designers of these languages took advantage of this capability - they embedded their languages in the underlying Lisp - but did not provide this capability to the users of object-oriented language. No documented mechanism supports definition of object language program elements which follow evaluation rules which are specializations of the standard ones. For example, a programmer cannot define a class with different rules for instance variable access and then mix that class freely with other standard classes.

CLOS is an extension to Common Lisp [5] which makes Common Lisp itself be an object-oriented programming language [6]. The integration of types and classes means that any Lisp data structure can be treated as an object. The integration of function calling and "message sending" means that generic dispatch is transparent to the caller. The metaobject protocol

ensures that the capabilities described above apply to the object oriented mechanisms in the language as well as the traditional Lisp mechanisms.

The Metaobject Protocol Provides Traditional Lisp Power

It is the existence of the documented metaobject protocol which allows CLOS to retain Lisp's capabilities to build and embed specialized languages. In traditional Lisp, this capability comes from the ability to manipulate program structure directly, and to extend the language by defining macros.

The metaobject protocol specifies that the the basic program elements are represented as first class objects called metaobjects [7][8]. These objects are instances of specified metaobject classes. Specified generic functions manipulate the metaobjects to provide the behavior of the system.

User programs can manipulate the metaobjects directly. This corresponds to the ability in Lisp to manipulate programs directly. By defining specializations of specified metaobject classes, users can define program elements that have behavior slightly different than the standard CLOS behavior. This corresponds to the ability to extend the behavior of the Lisp interpreter by defining macros.

Examples of the Power of Meta

The design of CLOS itself takes advantage of this ability to directly manipulate program objects. The initialization protocol described in chapter 1 of the specification [9] is an example of a higher level language construct built with more primitive CLOS program elements. The initialization protocol uses the ability to directly manipulate generic functions and methods to determine the full set of legal initialization arguments. This allows the initalization "language" to implement its own extension of the CLOS keyword argument congruence rules.

Since the Lisp community has had significant experience with declarative method combination, CLOS provides a specialized language for controlling

method combination. There is no corresponding language for describing how conflicts in slot descriptions are to be resolved (the default behavior is simple shadowing). The metaobject protocol makes it possible to change the default behavior, and Lisp allows the easy definition of languages to control the new behavior. Because users can extend the behavior of the underlying program structures directly, these specialized languages are not limited to what can be expressed with the base language.

The PCL user community has provided other examples of this power provided by the metaobject protocol. In this community many users have extended the standard behavior of the object system to suit their specific needs.

The Dangers of Standardization

One question which has been asked is whether it is wise to standardize on a language which includes new features, particularly an idea as new as the metaobject protocol. This is an important question and one which deserves consideration. It can be broken down into three smaller questions: will the standardized language be useful; can it be made to perform well; and will the standardization stifle research?

Will CLOS be Useful?

There are many reasons to expect that CLOS will be a useful language. It's basic object-oriented features are revisions of those found in Flavors, Loops and CommonLoops. The revision of these features has been done quite carefully, and tremendous attention has been paid to the comments received over more than 5 years from users of the previous systems.

The metaobject protocol only existed in CommonLoops previously, but already it has been used extensively. Even given its early and unsupported implementation stage, many large projects have switched to using PCL primarily because of the power the metaobject protocol provides.

Can CLOS be Implemented Efficiently?

Run time efficiency is dominated by the time to call a generic function and by the time to access the slots of an instance. Efficient implementation of these is complicated by CLOS features which support multiple inheritance, method combination, and the ability to change class definitions and have extant instances updated to the revised definition.

Techniques already exist for dealing with each of these problems, and several implementation efforts have been able to combine these techniques effectively. There are techniques for both stock and custom hardware. In addition, the metaobject protocol provides a sound footing for allowing the user to control special block compilation or "staticizing" techniques; there is promising research in this area.

An additional question is whether the existence of the metaobject protocol, with the extensibility it provides, prevents efficient implementation. While it is true that the existence of the metaobject protocol complicates the maintenance of internal datastructures such as caches, it does not interfere with the ability to use any of the existing or envisioned runtime implementation techniques. This has been demonstrated with PCL which has been used to experiment with a wide range of implementation techniques while retaining metaobject protocol extensibility. The existence of the metaobject protocol only requires that dependencies of each implementation technique be explicity represented.

A critical point to be made about CLOS performance is that it must be measured correctly. The ratio of generic function call time to ordinary function call time is often misused to measure CLOS performance. Since this ratio can be expected to be about 2 to 1, people often say that a CLOS program will be only half the speed of a comparable traditional Common Lisp program. This is not the case. A program written using CLOS is often likely to be faster than a comparable traditional Common Lisp program. To see this, it is important to understand what it means for a traditional Common Lisp program to be comparable to a CLOS program.

Generic function call time should be measured against not only the time to call an ordinary function, but also the time required to do the type dispatch (i.e. typecase) required to start executing the correct code. This is an appropriate comparison because the traditional Common Lisp program

has functionality comparable to the CLOS program.

The following two programs show corresponding programs written in CLOS and traditional Common Lisp. Of course the CLOS implementation retains greater flexibility since methods can be added to area without editing the original definition.

Ordinary Common Lisp:

```
(defstruct circle)
(defstruct square)

(defun area (shape)
  (typecase shape
    (circle ...code for area of a circle...)
    (square ...code for area of a square...)))
```

Common Lisp with CLOS:

```
(defclass circle () ())
(defclass square () ())

(defmethod area ((c circle)) ...code for area of a circle...)
(defmethod area ((s square)) ...code for area of a square...)
```

Will Standardization Stifle Research?

Perhaps the most important question is whether standardizing CLOS will stifle research. Rather, the opposite is likely to be true. The metaobject protocol exposes the underlying program structure. It allows programmers to experiment more easily with variations on that structure. Rather than stifle research, standardizing CLOS will make this vehicle for experimenting with Object Oriented Programming Languages widely available and so should encourage more widespread research. In the PCL community there has been more widespread experimentation with variations on the system than existed previously in the Loops and Flavors communities. These include small extensions, such as object naming mechanisms; medium sized experiments such as special slot inheritance mechanisms; and large exten-

sions like connections to persistent storage and large KR systems. The metaobject protocol provides a new way of looking at a standard. Unlike other OOL's such as Smalltalk [10] or C++ [11] it provides not just a current way of doing business, but a way to explore the future while sharing a common base.

References

[1] Matthew Halfant and Gerald Jay Sussman "Abstraction in Numerical Methods" *Proceedings of the ACM Lisp and Functional Programming Conference, 1988*

[2] Daniel Weinreb and David Moon. *Lisp Machine Manual* MIT AI Lab, 1981, Chapter 20

[3] David Moon. "Object-Oriented Programming with Flavors" *Proceedings of the ACM OOPSLA Conference, 1986*

[4] Daniel G. Bobrow and Mark Stefik. *The Loops Manual*, Intelligent System Laboratory, Xerox Corporation 1983

[5] Guy L. Steele. *Common Lisp: The Language*. Digital Press, 1984.

[6] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik and Frank Zdybel. "CommonLoops, Merging Common Lisp and Object-Oriented Programming" *Proceedings of the ACM OOPSLA Conference, 1986*

[7] Daniel G. Bobrow, Gregor Kiczales. "The Common Lisp Object System Metaobject Kernel, A Status Report" *Proceedings of the ACM Lisp and Function Programming Conference, 1988*

[8] Daniel G. Bobrow, Gregor Kiczales *The Common Lisp Object System Specification: Metaobject Protocol* X3J13 standards committee document 88-003, 1988.

[9] Daniel G. Bobrow, Linda de Michiel, Richard P. Gabriel, Gregor Kiczales, David Moon, Sonva Keene. *The Common Lisp Object System Specification: Chapters 1 and 2* X3J13 standards committee document 88-002R, 1988.

[10] *Smalltalk-80, the Language and its Implementation* Addison Wesley, 1983

[11] Bjarne Stroustrup. *The C++ Programming Language* Addison Wesley, 1986

Programming with Metaclasses in CLOS

Pierre Cointe & Nicolas Graube*

Equipe Mixte Rank Xerox France & L.I.T.P.
Université Paris VI
Tour 45-55 #209
4, Place Jussieu
75252 Paris cedex 05
France

This paper shows some possible uses of CLOS metaclasses at the enduser level as well as at the implementation level. Making metaclasses accessible to the user enables him to customize the system for his own applications. Using metaclasses at the implementation level makes the global task of describing the system as a uniform whole easier. We claim that CLOS metaclasses are necessary at both levels and will help us to suppress the boundary between endusers and implementors. In this paper we are not concerned with the efficiency of implementation issues. Our main concern is only to practice, understand and teach CLOS by using its oncoming meta-object protocol.

1 Programming with Metaclasses at the User Level.

In this section, we take two examples to describe some possible metaclass uses which can be conducted by a enduser in order to add new functionalities inside the system.

1.1 Abstract classes.

The first example is concerned with the definition of abstract classes as they can be found in Smalltalk-80 [9]. The purpose of such a class is to be used inside the inheritance lattice but not to be instantiated.

*This research was partly funded by the *GRECO de programmation du CNRS.*

Adding abstract classes to CLOS can be achieved by the definition of the new metaclass **abstract-class** and by the definition a new method for **make-instance**. All abstract classes will be instances of **abstract-class**. A simple way of avoiding the instantiation of these classes is to produce an error when **make-instance** is applied to them. Thus we just have to describe **make-instance** which produces the appropriate error.

```
;;; The metaclass.
(defclass abstract-class
          (standard-class)
          ()
          :metaclass standard-class)
;;;
;;; The method for make-instance.
(defmethod make-instance :before ((object abstract-class) ...)
   (error "This class ~S should not be instantiated"
          (class-name object)) )
```

1.2 Partwhole hierarchies.

As a second example we will introduce an implementation of Borning's Partwhole hierarchies [4] [1].

Partwhole hierarchies are a different way from inheritance of seeing composite objects. With this scheme, objects are built with various independent parts which are instances of some given classes. When the "whole object" is instantiated, its parts are automatically and recursively instantiated[1].

Only one metaclass, **Part-Whole-class** is necessary for the description of a partwhole hierarchy. This metaclass adds some new slots, thus permitting the description of both the parts and the associated classes of these parts. Classes which use parts should be instances of the metaclass **Part-Whole-class**. Methods should be added to generic functions involved in the class definition in order to transform parts into slot descriptions. The instantiation process of classes which include parts should also be modified in order to instantiate automatically the parts of each new instance.

```
;;; The metaclass.
(defclass Part-Whole-class
          (standard-class)
          (Parts Part-Classes)
          :metaclass standard-class)
;;;
;;; The method for initialize-instance
```

[1] A Smalltalk-80 implementation of partwhole hierarchies is discuss in [7]

```
(defmethod initialize-instance
        . :before ((object Part-Whole-class) ...)
    ...
    Transforms all parts into slot descriptions.
    ...)
;;; This method will generate the appropriate method
;;; for initialize-instance for the instance of part-class.
;;; It will perform the instantiation of the parts.
(defmethod initialize-instance
   :after ((part-class Part-Whole) ... )
  (with-slots (parts part-classes)
     (eval '(defmethod initialize-instance :after
              ((instance ,(class-name (class-of instance)))
                  &rest args)
               (with-slots ,parts
                   ,@(do ((parttparts (rest part))
                          (class parts-classes
                                 (rest class))
                          (result))
                         ((or (null part)
                              (null class))
                          (or result '(setf ,@result)))
                         (push '(make-instance ',(first class))
                               result)
                         (push ,(first part)
                               result)))))))
```

2 Programming with Metaclasses at the Implementation Level.

In this section we try to show that implementing Class taxonomy languages can be made easier by the use of metaclasses. We briefly present two attempts at designing a minimal kernel which must be able to support a full CLOS. Our goal is to identify the metaclasses which must define this kernel. Then in order to have a full CLOS we have to explain the new metaclasses and their relations in the class lattice.

As in [3] and [10], the basic idea is to describe CLOS itself as a CLOS programme with classes and methods.

2.1 From ObjVlisp to CLOS.

For several years we used ObjVlisp [5] [8] as a basis for experimenting with metaclasses. Although the main goal of this extension of Lisp was to explain Smalltalk-80 metaclasses, it has been used for the description of several other class-oriented paradigms. Naturally we tried to use it for the description of CLOS as an embedded system of ObjVlisp [6] [10]. One of the main ideas of ObjVlisp is minimality as reflected by its circular architecture which is composed of two classes only: `Class` and `Object`. We tried to work out these essential points of CLOS. This led us to point out three basic classes:

`CLOS-class` the standard class for all CLOS classes,

`CLOS-object` the class inherited by all CLOS classes,

`CLOS-slot-description` the class which describes slot objects.

Note that neither `generic-function`, `method` nor `method-combination` appear above because they can be built from these classes.

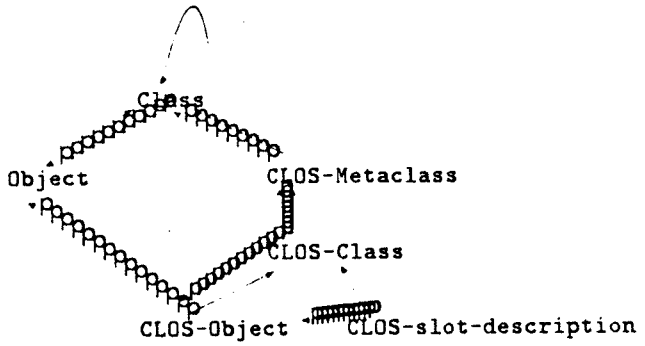


Figure 1: Inserting CLOS inside ObjVlisp.

The connection of these classes with the classes of the ObjVlisp kernel was accomplished by a specific metaclass which described some of the basic CLOS behaviours in order for `CLOS-class` to react like a class of CLOS.

However this was not completely satisfactory mainly because ObjVlisp makes some different assumptions from CLOS at the language level.

2.2 From μCLOS to CLOS.

From the previous description we outlined a reduced set of classes which can be used for a full description of CLOS compatible with [2]. Let us call this kernel μCLOS.

μCLOS is defined by three classes:

standard-class the standard class for all μCLOS classes, this class is self-instantiated.

standard-object the class inherited by all μCLOS classes,

standard-slot-description the class which handles the slot description.

A first step towards the description of CLOS is the connection of the class lattice with the underlying type lattice. This relation is accomplished through the metaclass **Type**. Then **T** becomes an instance of **Type**. The definition of **T** as a class implies the modification of the inheritance of **standard-object** which now inherits from **T**.

Generic functions, methods and method combinations can be added to this kernel by the simple description of their related classes, **generic-function**, **method** and **method-combination**.

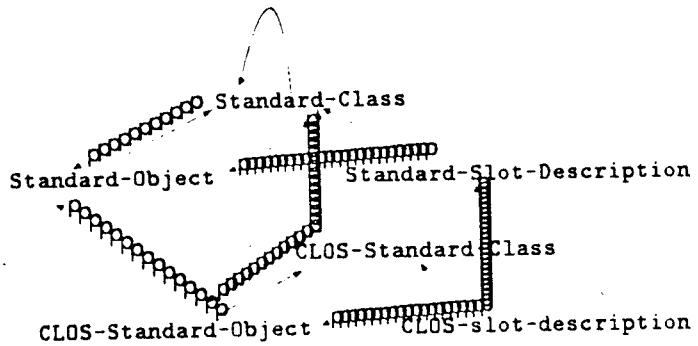


Figure 2: μCLOS extented to CLOS.

At this stage we hold all the basic functionalities required for the description of a complete CLOS. This task could be made easier by the existence of our μCLOS. It could be achieved through a new inheritance lattice of specific classes connected to that of μCLOS.

3 Conclusion.

Metaclasses are useful for anyone who tries to customize the system. Using a meta-object protocol gives access to any component of the system thus allowing both its understanding and modifications. Since CLOS has become the standard object system for Common Lisp it is high time to teach it. We think that in order to understand all its new - and often complex - mechanisms, an approach developing the CLOS metaclass architecture will make this task much easier.

References

[1] Blake, E., Cook, S., On Including Part Hierarchies in Object-Oriented Languages, with an implementation in Smalltalk. *ECOOP'87*, Lecture Notes in Computer Science, Vol 276, pp 41-50, Paris, France, Juin 1987.

[2] Bobrow, D.G., DeMichiel L.G., Gabriel R.P., Keene S., Kiczales G., Moon D.A. Common Lisp Object System Specification, Chapter 1 and 2, X3J13 (ANSI COMMON LISP), November 1987.

[3] Bobrow, D.G., Kiczales G., The Common Lisp Object System Metaobject Kernel: A Status Report, *Lisp and Functional Programming 1988*, Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, pp 309-315, Snowbird, Utah, July 1988.

[4] Borning, A., Duisberg, R., Freeman-Benson, B., Kramer, A., Woolf, M., Constraint Hierarchies, *OOPSLA'87*, Special Issue of SIGPLAN Notices, Vol. 22, No 12, pp. 48-60, Orlando, Florida, USA October 87.

[5] Briot, J-P., Cointe, P., A Uniform Model for Object-Oriented Languages Using The Class Abstraction, *IJCAI 87* Proceedings of the Tenth International Joint Conference on Artificial Intellingence, pp. 40-43, Milan, Italy, August 1987.

[6] Cointe, P., Towards the design of a CLOS Metaobject Kernel: ObjVlisp as a first laver, *IWoLES 88*, Afcet, Afnor, LITP and Inria, Paris, France, February 1988.

[7] Cointe, P., A Tutorial Introduction to Metaclass Architecture as provided by Class Oriented Languages, *International Conference on Fifth Generation Computer Systems*, Tokyo, December 1988

[8] Cointe, P., Metaclasses are First Class: the ObjVlisp model, *OOPSLA'87*, Special Issue of SIGPLAN Notices, Vol. 22, No 12, pp. 156-167, Orlando, Florida, USA October 87.

[9] Goldberg, A., Robson, D., Smalltalk-80 - The Language and its Implementation, Addison-Wesley, Reading MA, USA, 1983.

[10] Graube, N., Reflexive Architectures: From ObjVlisp to CLOS, *ECOOP'88*, Lecture Notes in Computer Science, Oslo, Norway, August 1988.

[11] Maes, P., Concepts and Experiments in Computational Reflection, *OOPSLA'87*, Special Issue of SIGPLAN Notices, Vol. 22, No 12, pp. 147-155, Orlando, Florida, USA October 87.

[12] Queinnec, C., Cointe P., Types, Classes, Metatypes, Metatypes Classes: an open-ended data representation model for Eu-Lisp *Lisp and Functional Programming 1988*, Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, pp 298-308, Snowbird, Utah, July 1988.

[13] Steele Jr., G., F., Common Lisp: The Language, Digital Press, 1984.

Contents

Translation of KEE Object Oriented Programs into CLOS-XT

Position Paper

CLOS Workshop, October 1988

A. Dappert-Farquhar
J. Estenfeld
ZTI INF 31
Siemens AG
Otto-Hahn-Ring 6
8000 München 83
FRG

0. Abstract

In recent years, at Siemens we have developed a number of applications using a subset of the object-oriented programming facilities of KEE. Currently it is our group's intention to use CLOS for the development of future in-house applications. In order to do so, we are now working on an extension of CLOS, called CLOS-XT, to suit our specific application development requirements and also on a transformer which automatically translates the object-oriented parts of existing KEE knowledge bases into CLOS-XT.

CLOS-XT = CLOS Extension using XT

1. Extensions to CLOS

Our CLOS extensions, called CLOS-XT, arise out of the specific requirements of the applications to be developed: In order to assist users who are used to KEE UNITS facilities in getting acquainted to CLOS we add a number of helpful properties to CLOS. At the same time this makes it easier to transform existing KEE knowledge bases into this extended CLOS. Some of the extensions are the following:

- In addition to the CLOS defined slot options we offer an inheritance option, which allows inheritance to be terminated and it is also our intention to provide the possibility of choosing a user defined form of inheritance . There is also a documentation option for each slot.
- We extended the slot options in *defclass* to not only allow for system defined options but also for the specification of further user defined annotations. These are described by the possible annotation options *name*, *initform* and *documentation*.

- Instances can be named objects. Every class 'knows' its instances.
- Classes and instances are assigned to and administered by knowledge bases.
- In addition we intend to offer a variety of functions in order to manipulate classes and instances (like removing classes, changing a class's knowledge base, adding a new superclass, etc.) or to obtain information about them.

2. Transforming KEE Knowledge Bases into CLOS-XT

Switching from the object-oriented programming facilities of KEE to CLOS-XT makes it necessary to convert existing KEE knowledge bases. This consists of two tasks:

2.1. Transforming KEE knowledge base files into CLOS-XT files
2.2. Transforming KEE functions

2.1. Transforming KEE Knowledge Base Files into CLOS-XT Files

The main problem here is the conceptually different view taken on
- KEE units and CLOS classes and instances
- KEE own and member slots and CLOS instance slots

At present we are implementing an algorithm which transforms the list structures describing units into definitions of knowledge bases, classes and instances. By parsing *superclass.parent.list*, *member.of.parent.list* and *member.slot.list* a distinction can be made about whether one is dealing with a knowledge base, an active value, a unit that can be treated as an instance, or a unit that is to be treated as a class. Depending on the presence of *own slots* and their inheritance, artificial meta-classes need to be created.
KEE methods are not attached to slots of the respective classes but are transformed into CLOS methods; active values become methods.

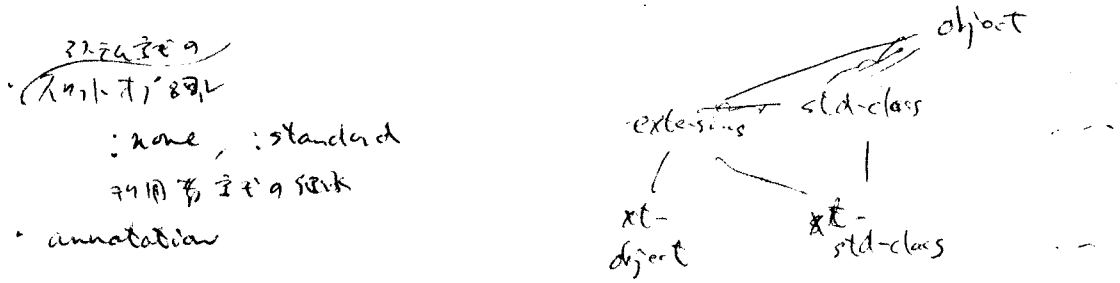
2.2. Transforming KEE Functions

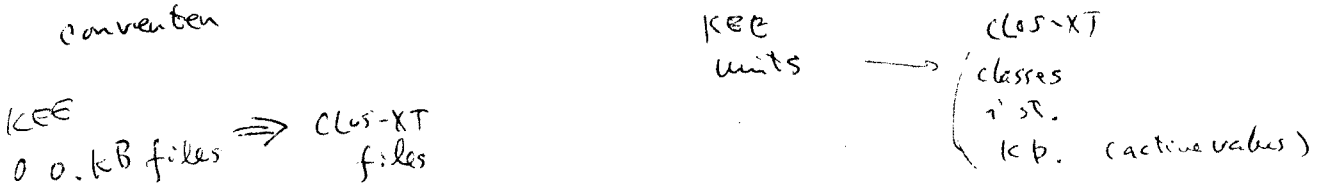
Here again there are KEE functions referencing units which have to be transformed into functions with different behaviors for classes or instances. This is determined by parsing the functions' parameters. We only transform functions

that are relevant at run-time and therefore are to be expected in KEE knowledge bases. Work on this module as yet has not begun, and therefore it is hard to forsee how well this can be done as it might prove difficult to model the correct dynamic behavior from the available information.

3. Conclusion

CLOS-XT is being realised on the meta object protocol level, the transformer is being implemented in CLOS-XT. It seems that a fully automatic transformation is not possible and that not all KEE functionality can be transformed cleanly without changing most of the CLOS properties. This is not reasonable and also not necessary. We therefore had to impose some restrictions on current developments with KEE UNITS, and are forced to mark places in the transformed files where manual corrections might be needed. Examples for these restrictions are not to allow multiple class parents for an instance (i.e. multiple *member parent links*) or restricting slot values to single values. We nevertheless expect the automatic transformation to be a worthwile support and hope that it will assist users in getting acquainted with CLOS-XT. CLOS-XT and the transformer are not planned to become company products, but are intended to be used for internal purposes only.





CLOS Workshop Position Paper

John R. Davidson
H. James Antonisse
Richard W. Tucker
MITRE Corporation
Washington Artificial Intelligence Technical Center
davidson@mitre.arpa

September 14, 1988

Abstract

MITRE's role as a Federal Contract Research Center involves the MITRE Washington AI Center in the specification of DOD systems acquisitions and the rapid prototyping of specific DOD applications on diverse hardware configurations. This role requires us to review currently available expert software packages as well as develop applications from either in-house or public domain software bases. In the latter effort, the use of a common software base to facilitate quick gearing up for prototypes is essential, but is complicated by the need to support a diverse range of underlying hardware suites, windowing, and graphics display systems. CLOS addresses a major part of these issues.

MITRE is seriously interested in CLOS as a vehicle for moving application-layer software across hardware boundaries in a timely manner. The emergence of CLOS as a underlying object representation language and combined with a common hardware driver (as in the X windowing system) in CLUE/CLX provides an attractive and integrated solution to linking semantic networks, graphical displays, and other user interface presentations in portable expert system applications. "Extensions" to CLOS in the form of meta-level reconfigurations to implement contending inheritance strategies and datatype definition facilities is anticipated.

1 Semantic Networks and Rule Interpreters

Many of the MITRE prototype expert systems employ semantic networks in one form or another. Their actual representations have in the past been driven less by the functionality required for an inheritance system than by the representation given by the specific rule interpreter chosen for the effort. Some of our rule languages are built on top of FRL, others on FLAVORS, some DEFSTRUCTS. The MITRE Washington AI Center has a new software support task to develop a general-purpose inheritance system written in Common Lisp for portability. CLOS is an attactive candidate for the implementation language.

2 User Interfaces

We will also be extensively engaged in developing an interface to the inheritance system. We want to avoid the need to have machine-specific user interface specialists for systems prototyped on either the Symbolics, SUN, MAC-II, Explorer, etc., (Sorry Gregor, we don't have any XEROX systems). Provided all our major hardware candidates support some form of Common Lisp and X (admittedly a big assumption), a user interface language built on CLUE would serve us extremely well, perhaps reducing prototyping time by as much as a half. In addition, new capabilities developed during one effort would more easily port to the next. We expect considerable effort to be spent on CLOS and CLUE in this regard.

3 Graphics Displays

We have an in-house graphics display subsystem called MMI that presently runs solely on the Symbolics (it used to run on the LMI Lambda as well). The package performs some extremely useful coordinate system mapping and display tasks, and has been used widely at MITRE for prototyping dynamic situation displays. It is a large system requiring near constant care and feeding by a Symbolics wizard. Several of the original designers and implementers have since moved on, resulting in periodic trauma as new bugs are introduced and features are eliminated. We really need this system to be based on a device-independent windowing system, such as X. In addition, the maintainability of the MMI system would increase tremendously as the underlying software is no longer home-grown, but a product of a wider

CLUE

debate in AI community, with its own user base and bulletin board discussion forum.

4 Strong Datatyping

Our work on a database definition facility stands at the intersection of CL, CLOS, and CLUE. It is based on a menu-driven control of database composition, where those types are entered in the inheritance system and potentially reasoned about.

5 Summary

The above are major areas of interest in CLOS at MITRE, all of which seem to be addressed to one degree or another by the CLOS and CLUE developments currently in progress. We have been using PCL as a provisional tool for nearly a year, and have constructed a small classification system using PCL as the supporting object language. The MITRE Washington AI Center will be investigating these tools in a software support effort over the next year or so, and we will be attempting to develop an integrated set of device-independent tools for developing semantic networks, graphical displays, datatyping languages, and general user interfaces for the rapid prototyping environment.

MICE: A Modular Intelligent Constraint Engine

Position Paper
CLOS Workshop 10/3-4/88

Harley Davis
Sanjay Mittal
Xerox PARC Systems Sciences Laboratory
3333 Coyote Hill Road
Palo Alto, CA 94087
hdavis.pa@xerox.com
mittal.pa@xerox.com

Introduction

MICE is a tool for defining constraint satisfaction problems and solving them using a modular intelligent search engine. The search engine is modular in that it allows the specification of a set of intelligent search techniques to be used within a basic framework to generate a specific problem solving algorithm. In addition, MICE supports the use of a variety of representations for constraints, variables, and domains. MICE is written in Xerox CommonLisp and PCL; the user interface relies additionally on the Xerox Lisp window system.

More Detail

A constraint satisfaction problem (CSP) consists of a set of variables each associated with a domain and a set of constraints on these variables. Problem solvers for CSPs on serial machines generally take the form of a backtracking algorithm. Research on CSPs tends to focus on particular problem solving strategies for such problems, often assuming a particular domain or constraint representation and a specific type of problem. MICE allows the integration of these strategies into a general backtracking framework by defining a number of problem object classes each using a different representation, but sharing a minimum protocol.

Each of the problem objects, such as a domain or a constraint, is selected from a set of classes. The protocol defined on the root node of the class graph is applicable to all object representations and can be relied on by any solution strategy; more specialized representations might define more specialized protocols which only work with a limited set of solution strategies. The problem-solving strategies, represented in the current version as methods, themselves take different forms depending on the particular combination of problem object representations used in them; CLOS multi-methods are particularly useful in this capacity.

We also define a solution specification object, whose various slots specify different parts of an overall solution strategy. The general problem solving algorithm is then filled in by the pieces in the solution specification object. The class of the solution specification may even implicate a more specialized problem solving algorithm when the combination of methods is known to be reducible to a more efficient algorithm. For example, simple chronological backtracking requires less state information than used by other intelligent mechanisms; we can use a simpler general framework for the class of chronological solution specifications.

This leads to certain peculiarities in the object graph. Each slot in the solution specification object has a number of choices, most of which can work with a large combination of each of the others. One way to represent this state of affairs would be to provide mixin classes for each choice of each slot and define problem-solving methods on all combinations of these mixins. However, this would result in an explosive growth of classes and methods - most very similar - and would be largely unmaintainable. At the other end, we could provide a very generic problem solving strategy and not maintain subclasses of the solution specification, relying on the user to fill in all slots of the solution specification before running the problem solver. This approach fails to take advantage of the natural synergy of certain combinations of slots in the solution specification.

The approach we have chosen falls somewhere in between. We define a completely generic problem solver, and provide a small set of subclasses to the most generic solution specification on which more specialized problem solving methods are defined. Any of the slots in the solution specification may be changed by the user, but the more specialized problem solvers may not be able to deal with any combination. This conflict between a large lattice of mixin objects and their combinations, with many specialized methods, and a small graph of objects with a more general but more inefficient interpreter is likely to occur often in object-oriented programming; adequate solutions depend largely on the particular domain.

There is often considerable interaction between the various modules of the problem solver; several are complementary consumers and generators of problem state. Thus, we define a search state object which can be used for communication between the modules. The search state object can be subclassed for specialized and closely interacting problem solving modules. This solution is not ideal. Ultimately we would like to reduce the reliance on one monolithic state object by defining larger-grained problem solving modules which would package together generated and consumed state and thus obliviate the need to define this state when the modules aren't being used.

The MICE interface class graph closely parallels the problem solving object class graph. An interface unit consists of a problem definition window and a problem solution specification window (and some control menus). The problem specification window contains cells for each domain, variable, and constraint in the problem; these display a small amount of useful information for each object. The cells can be expanded into problem object editors; the format of the editor depends on the class of the problem object. A domain editor differs from a constraint editor; an extensional domain editor differs from a range domain editor. The problem solution window allows the selection of the set of strategies to be used in the next problem solver execution. All of the display objects are connected to their referent problem objects, but do not interfere with them; thus, we can develop multiple MICE interfaces.

CLOS Issues

In a search problem (as in movie production), performance is the Paramount issue. Optimization of the various modules and overall problem solving framework has left a considerable computational load on PCL. Although increasing PCL performance will only result in a constant speed increase and is thus not theoretically interesting,

realistic use of the system as a unit in some application demands high performance. Thus, we would like to see performance issues stressed.

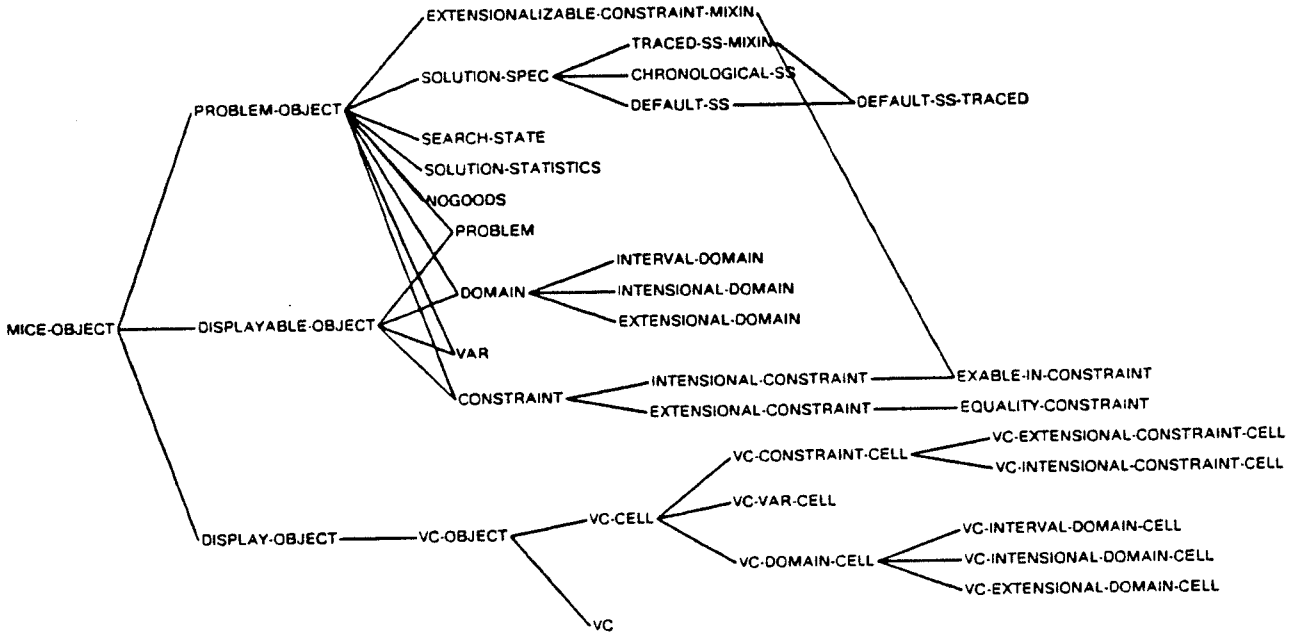
The metaobject protocol is also of interest. We define classes of problems; subclasses of problems inherit problem objects (constraints, variables, and domains). Currently we must support this behavior through complex initialization procedures; this might better be done by defining problem meta-objects.

Certain aspects of the CLOS specification are found to be frustrating in practice. For example, it is often useful to specialize on optional arguments; this might be integrated into CLOS without violating the notion of congruent lambda lists.

Finally, as a programming effort MICE has often been frustrated by the lack of environmental integration in PCL. Although we realize this issue is separate from the actual CLOS specification, this workshop will be a useful forum for discussing the kinds of tools CLOS programmers need to work effectively.

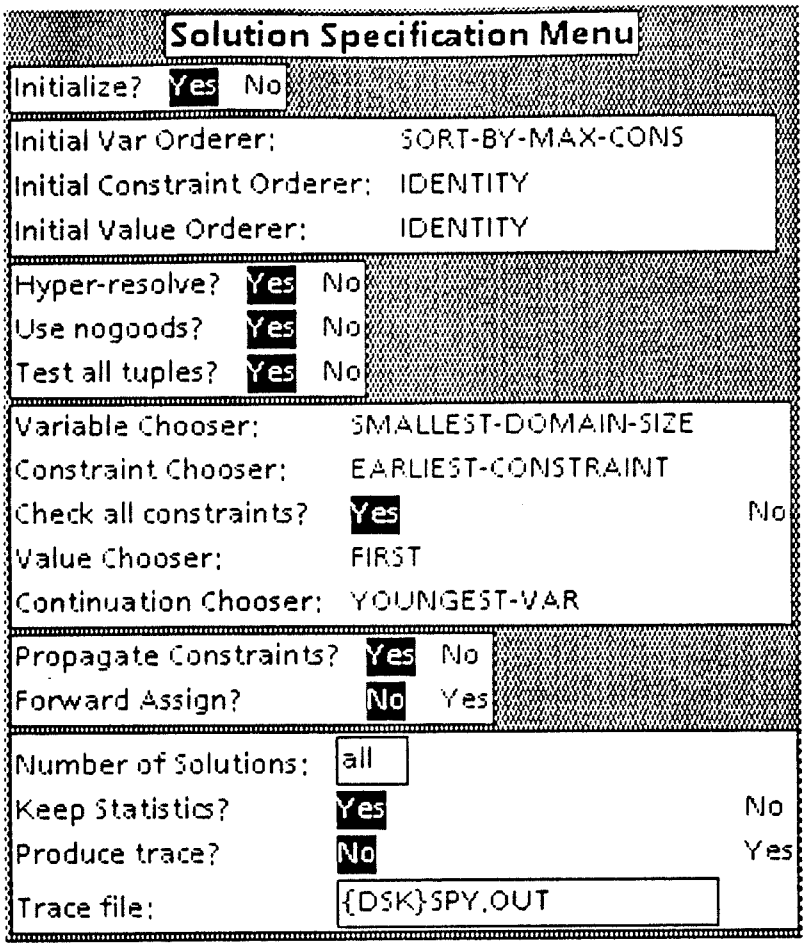
Appendices

1. Part of the MICE Class Graph

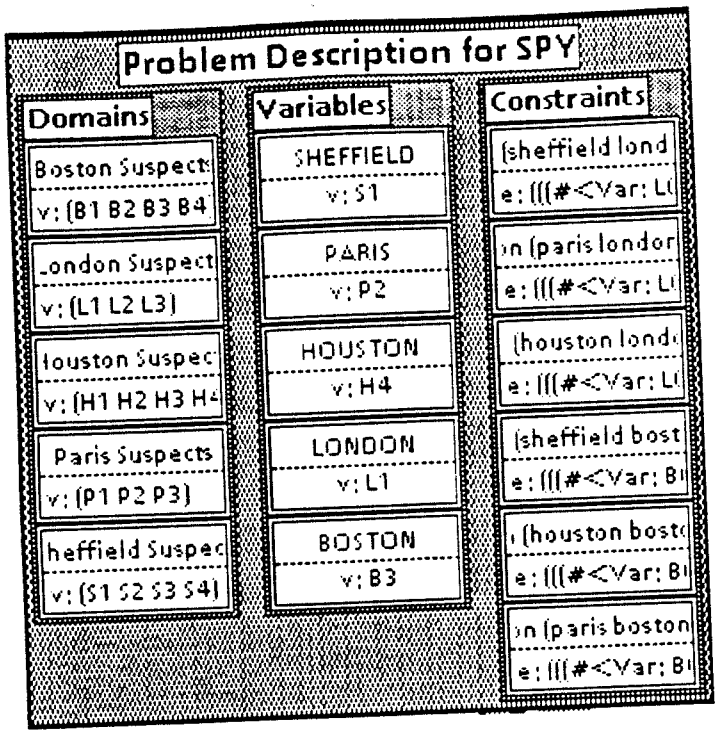


2. A snapshot of the MICE interface

a. The solution specification menu



b. The problem description menu, with cells for domains, variables, and constraints.



CLOS issues

- multimethods
- Method Combination
- Mixin Madness
  Exponential growth of possible solution specification
- Going Meta
- Communication objects

この図は CLOS issue の中の mixin madness の事例.

CLOS Position Paper for Digital Equipment Corp.
==========

Meta-Objects for Efficiency

As a Common Lisp vendor, we have an interest in insuring that Common Lisp be as time- and space-efficient as possible. Although CLOS Chapters 1 & 2 have already been adopted by X3J13, we are keenly interested in the development of the remaining chapter(s), and making sure that efficiency goals are not inadvertently subverted. In particular, we would like to explore how the richness which CLOS provides for development can be subsequently restrained (presumably via the meta-object protocol) for the purpose of delivering a debugged application which is as small and fast as possible.

Implementation Issues

More pragmatically, we are interested in examining CLOS implementation issues, especially as they relate to efficiency of application code. We would be interested in a discussion of the (obviously significant) engineering behind PCL. We are also concerned about bootstrapping issues, which make having delivered systems more complex than perhaps is necessary.

PCL Status

A status report on PCL would be a welcome topic, as would any discussion of future plans and (even tentative) schedules.

Debugging Tools

Providing good debugging tools in an environment is an important issue. We would like to learn about ways of displaying (graphically or otherwise) the interesting parts of the computation, including how various applicable methods have been run or are about to run, or how particular methods weren't run. Also providing graphical ways of manipulating classes should not preclude any significant functionality available programmatically, and should offer the ability to translate to and from source code.

Concurrency

We would also like to see CLOS address issues of concurrency in a reasonable fashion. Although, strictly speaking, concurrency issues exist only outside the universe of Common Lisp, and hence CLOS, discourse, the use of multiprocessing systems supporting either shared memory or network communications has become common, and concurrency support in an object-oriented system will be one of the keys to acceptable applications performance. This support might range from that which provides for object-internal concurrency with appropriate synchronization and notification primitives, to that which merely provides for object-external concurrency by requiring a true message-passing method invocation interface. Consideration must be given to whether these objects are viewed as "active" or "passive" in their relationship to threads - whether an object encapsulates (always) a

thread which executes all methods of the object, or whether any thread can invoke methods of any object and execute them in its own context (on its own stack).

Persistence

Coming in the near future is the need for persistent data in the object-oriented environment. And persistence may have a significant impact on the concurrency support offered, with the need to support multiple-object transactions of both long-term and short-term duration.

Integration with Other Systems

Although CLOS represents a new standard for object-oriented systems, we are interested in issues surrounding the integration and interaction of CLOS with other, pre-existing, object-oriented systems. Especially of interest to us are CLOS interfaces to various graphics and windowing systems, including X, CommonWindows, and others. Also, we would like to hear about any work that has been done to try to integrate CLOS with other object-oriented languages or language extensions, such as C++. Just as many vendors provide access to other programming languages from Lisp, we anticipate a demand for interfaces between CLOS and other object-oriented languages.

TICLOS: A high performance implementation of CLOS for the Explorer family

Patrick H. Dussud

September 9, 1988

1 Goals

The following goals were set at the beginning of the design:

- Provide a complete implementation of CLOS, as defined in X3J13.
- Provide a commercial high performance implementation of CLOS. A measure of success will be that a piece of code converted from Flavors to CLOS runs at least as fast as it ran under Flavors.
- Provide low level data reprentation compatible with Flavors. This avoids modification of the memory management system software, and various other related utilities.
- Provide high level compatibility wtih Flavors. At least, Flavors should be some sort of CLOS classes and it is possible to specialize a CLOS method on a Flavor class.

2 Design Philosophy

Given the above goals, the design CLOS was based on our experience with the Flavors system. The design gave priority to Flavor like operations:

- Fast runtime execution, at the expense of method and class creation, or modification, by precomputing all the information that will be needed at runtime.
- Fast single argument method discrimination. Multi method and individual methods discriminations are done in multiple steps of the basic mechanism.
- Fast instance access, at the expense of more data structure overhead.

Another design goal was to limit the amount of special support in microcode to a minimum: The system should be implemented in Lisp, and not in microcode, unless there is a benefit to get from the hardware. The amount of special support is significantly lower that the one devoted to Flavors. The advantage to this design philosophy is that it provides us with a solid starting point: We can get performace statistics from Flavors; The simulation of the design can be tested on data gotten from real program measurements.

3 TICLOS implementation

3.1 Representation of instance and classes

3.1.1

Standard-class objects are split in two parts. The class object points to a class-description structure that contains the information accessed through the standard chapter 3 accessors. This allows for Class redefinition.

3.1.2

Instances have a header word, containing a tag and a pointer to their class descriptions. After the header word, comes all of the instance slots. This is very similar to Flavors, where the class-description structure is called Flavor structure. This enables us to run without modification to the garbage collector system. TICLOS instances are distinguished from Flavor instances by a bit set in the header word (most significant cdr code bit).

3.2 Representation of generic functions

Generic functions are represented as a normal compiled function (FEF), a debug info slot points to the data-structure containing their slots. This allows for fast generic function call with little support from the microcode.

3.3 Representation of Methods

Methods are represented as instances. One of their slots points to a compiled function(FEF) that implements the methods.

3.4 Method discrimination

Method discrimination is done by calling the microcoded routine (miscop) %dispatch-method. This routine accesses the generic function arguments, and uses the pointer field of the instance header (pointing to the class description of the instance) of the most significant argument as a key to a dispatch hash table. The value will either be an effective method to be called, or another hash table if other discriminations have to be performed. When the effective method is called, the call is tail recursive: The call frame of the generic function is updated, and re-used for the method call frame. This makes sense because the effective method receives the same arguments that the generic function received. Benchmarks show that when a generic function is called, and one argument discrimination is performed, TICLOS is faster than Flavors' SEND.

3.5 Slot access within a method

The optimized instance access uses mapping table technique analog to Flavors. The implementation is extended for the case of multiple instances access inside of a method: There is one mapping table passed for each specializable argument. These mapping tables are passed directly by the call instruction to the methods, via specialized locals. This avoids rearranging the stack to accomodate for more arguments than the one originally passed to the generic function. The mapping table mechanism has been extended to handle access to class slots, and dynamic class redefinition. The mapping table stored values can be one of:

- A fixnum, when the slot is an instance slot. It codes the position of the slot in the instance.
- A locative, when the slot is a class slot. It points to the class slot location, inside the class-slot structure.
- Some symbolic values, to code the situations, where an instance is obsolete, or when a slot does not exist anymore.

The microcode following mapping table indirections uses the Explorer specialized hardware to branch through these cases without cost.

4 Conclusion

TICLOS implementation, on the Explorer architecture led to a system, that outperforms Flavors, and increases the functionality of object oriented programming significantly. Even though it supports the most sophisticated features of CLOS, (instance update after class redefinition, and multi-methods), careful design led to a no performance penalty for the common case. We anticipate a change in programming style due to the powerful new features of CLOS (multi methods, and individual methods, metaclass programming). This will lead to some additional tuning of TICLOS, as performance data is available.

References

[Moon 86] David Moon, Object Oriented Programming with Flavors, OOPSLA' 86.

[TI] Explorer System Design Notes, TI Part Number 2243208-001*A.

SHARKII

John Dye
IU Lab Director
Advanced Decision Systems

This letter is my submission of a position paper for attendance of the Clos Workshop at Xerox this coming October 3-4, 1988. At the workshop I will represent Advanced Decision Systems whose research projects make extensive use of CLOS. In my position as Director of the Image Understanding Laboratory (IU Lab), I use and support two large software systems (SharkII and View) which are built upon the CLOS object-oriented language standard. My purpose for attending the CLOS workshop is twofold: 1) to provide the perspective of an application author with experience in using CLOS to implement large systems and 2) to engage in discussion about the suitability of CLOS for developing large time-critical applications in LISP.

My perspective as an applications author stems from over a year of experience using the PCL software in implementing two large systems: SharkII and View. SharkII is a User-Interface toolkit written in CLOS and COMMON-LISP for the Sun Workstation which was demonstrated at AAAI-88 at the Lucid booth. SharkII is built on the NeWS windowing system from Sun and provides an object-oriented interface to lisp programmers using NeWS. In addition to the standard user-interface objects, SharkII provides several high-level support packages including Image display, Chart Drawing, Table Interactions and Graph Interactions. View is ADS's representation language for describing spatial objects and processing over them. Written in Clos, View makes extensive use of inheritence and method combination. View provides macro constructs for rapid access and iteration over these spatial objects. In addition, View provides an object-oriented database capability to relate the objects with each other. View also provides the ability to display the Spatial objects using the Shark user-interface constructs. As part of attending the workshop I would be pleased to demonstrate the SharkII user-interface running on the Sun4/110c. SharkII requires about 25 megabytes of disk space for the software (which includes one 10megabyte lisp image). In addition SharkII requires the Sun NeWS version 1.1 software be installed. SharkII also requires at least 50 megabytes of swap space be available to both NeWS and Lisp.

Lucid Beta 5

My experience with CLOS on both the View and SharkII efforts has led me to a belief that the CLOS implementation must be very efficient if it is to be used for user-interface and image understanding applications. I believe that CLOS must be efficient in the instantiation of objects and in doing method combination for it to be widely accepted. Often applications like computer vision or user-interface require thousands of objects to be created at once (e.g. when doing an edge-extraction or when displaying a large Table). The time and space used by CLOS in making large numbers of object instances is crucial to the performance of these applications and to the acceptance of CLOS by programmers who work in these areas. I am also interested in tequiniques to model the amount of performance penalty of using CLOS methods over Common-Lisp defuns. It is important that this penalty be a small fraction of the time spent in function execution and that this penalty grow sub-linearly with the number of objects defined in CLOS.

I believe that the real-world experience I've had with CLOS will provide a useful perspective at the workshop. In addition, I am excited about the possibility of becoming more informed about the theoretical aspects of CLOS, particular Meta-Objects.

Database Support for Object Oriented Programming

Steve Ford

Information Technologies Laboratory
Texas Instruments Incorporated
P.O. Box 655474, M/S 238
Dallas, Texas 75265
(214) 995-0362
ford@csc.ti.com

ABSTRACT

Zeitgeist is an Object-Oriented Database (OODB) System being built by the Information Technologies Laboratory of Texas Instruments in support of programming environments. In addition to providing the traditional benefits of database support, stable long-term storage and controlled sharing of information, Zeitgeist is integrated into the programming environment which utilizes it. Our premise is that the data structures and data manipulation primitives provided by computer programming languages afford the most natural interface to data regardless of the location of that data (i.e. local memory or remote database), and that database data models and their associated specialized query or data manipulation languages tend to be more an artifact of the design of databases than a natural interface to them. The single deficiency of most programming languages in assuming the role of database languages is their lack of a provision for dealing with the temporal dimension. Rather than to invent yet another language, we have chosen to extend popular programming languages in this direction, starting with Common Lisp. Our current implementation permits a programmer to designate any normal transient Common Lisp data object or Flavor object as database resident. All subsequent references to that object, the objects it references, or future versions of them, will be transparently redirected to the database. The programmer continues to manipulate the object as if it was resident in virtual memory. This seamlessness between transient and persistent data has trivialized the conversion of several applications from virtual memory implementations to database implementations.

Experience has shown that the persistent object of choice is the Flavor object. The advantages of the association of a data structure with the operations that manipulate it is only magnified when that object, its definition, and its methods actually live outside of virtual memory, and are shared among users on different machines. In Zeitgeist, no explicit action is required to maintain that association in the database. The biggest disadvantages of Flavors is its relative lack of portability and extensibility, problems addressed by the CLOS standard and its Metaobject Protocol.

Shortly, we will be adding CLOS support to Zeitgeist, and porting our system to UNIX platforms. We are interested in the general experiences of others who have extended CLOS, and, more specifically, in a number of persistence and temporal language issues that have arisen from our integration of an OODB with a programming language.

For instance, Common Lisp semantics don't allow two symbols with the same name to exist concurrently. How do we reconcile semantically the ability to manipulate two versions of a symbol simultaneously in memory? The same issues apply to packages, structure definitions, and CLOS class definitions.

Also, CLOS requires that changes to a class be propagated to its instances. Does this requirement disallow the simultaneous manipulation in memory of different versions of an instance created with different versions of a class?

We make use of the low-level invisible pointers and trap handlers common to most Lisp implementations to identify persistent objects and slots, and to transparently fault in data from the database. Would it be desirable to formally define these widely-used mechanisms, and add them as language features?

Our implementation defines an external data representation and the associated translation and data compression routines for Common Lisp, as do other similar implementations. Is it appropriate to converge on a common interchange format for the language?

We look forward to more discussion of these issues. For more information on Zeitgeist, a detailed report on this project appears in the Proceedings of the Second International Workshop on Object-Oriented Database Systems.

Zeitgeist = O·O データベース の プロトタイプ版 を 作り

Position Paper for Franz Inc
PCL workshop.

We are currently using PCL for writing window-based debugging tools which include an inspector, process browser, stack browser, parameter browser, profiler and grapher. Our goal is to not only provide these tools but write them in a way that they can be modified and extended by users. We've been very pleased with the ease at which code can be developed and debugged with PCL. Naturally we are also working on reducing the overhead for generic function calls. Our efforts are currently machine (and Allegro Common Lisp) specific.

For the future we see ourselves working in three directions: One is to provide more tools written in PCL. Second is to continue to improve the speed of generic functions calls and instance variable references. Third is to turn inward and rewrite parts of the Lisp system in PCL so they can be easily extended by users.

The following issues concern us as developers of products in PCL and as people supporting customers of our Lisp who use PCL:

What is the timetable for PCL development?
When will machine independent speedups (e.g. method lookup) be added to PCL (if they will be added at all)?
What other changes are anticipated?

How long will PCL work be done at Xerox and freely given out? Will the Xerox lawyers decide that they have something valuable on their hands and start charging a license fee for the code? If they have no current plans to charge such a fee will they make a written statement they will never charge such a fee?

The fact that there is one official set of sources for PCL has made PCL code very portable across implementations. It is frightening to think of what would have happened if various Lisp implementors were just given the CLOS spec and told to implement that. No document can be as unambiguous as the sources themselves. In the future as PCL is imbedded in various systems people are going to be forced to diverge from the official source and it is important that the management of the sources permit implementations to diverge in different and substantial ways.

We would like there to be some way to separate the parts of PCL just needed for the execution of PCL methods from the parts needed to define new classes and methods.

Are there plans for a test suite for CLOS?

For our system we would like to be able to define everything in the PCL syntax and then easily trade speed for extensibility by declaring that certain parts of the class hierarchy are frozen.

The people at Franz who are involved in teaching Lisp have the following observations to make.

Until very recently object oriented programming was very closely associated with Smalltalk, Lisp, and for those who knew about it, Simula. The fact that these languages were themselves considered somewhat exotic and not readily available to most programmers added to the mystique which surrounded object oriented programming.

The recent interest in oops in the popular computer press, the use of oops in data base technology, and the development of C++ have increased programmer understanding of what the term means. For some, it has provided the opportunity to use object oriented programming within a familiar programming framework.

In addition, numerous tutorials on Object Oriented Programming are now advertised. While I have no data on the response to these ads, it is clear that training of this sort is available to those who want it.

In consequence of these developments, the task of training, and to a lesser extent documentation, of CLOS can concentrate more on the features of CLOS rather than providing the explanation and experience necessary to deal with the basic ideas of oops.

Furthermore, because many of the magazine articles and tutorials concentrate on applications with which a non-Lisp programmer may be familiar, the task of a CLOS instructor includes showing how CLOS is really an extension of ideas and techniques which are prominent in other aspects of Common Lisp.

The increased sophistication of the general programming public with respect to object oriented programming also demands that people teaching CLOS be much more knowledgeable than were their predecessors who taught other object oriented programming systems which were extensions to Lisp. In the past, anyone with access to an implementation of object oriented programming, and its documentation, could very easily be more knowledgeable than very sophisticated programmers who had no access to these tools.

Some Remarks on the Common Lisp Object System

Richard P. Gabriel
Lucid, Inc. and Stanford University

In other forums I have argued that programming language standardization freezes advancement and research on the standardized language, if for no other reason than the fact that funding for research on a standardized language is hard to get. The more advanced and fertile the language the more the pity when it is standardized. Moreover, standardizing a language whose design is significantly beyond that of currently used languages is risky if the benefits of experience have not been folded back into the language.

The Common Lisp Object System represents a relatively large departure from known languages to be incorporated so quickly into a language standard. The Object System uses **method applicability** instead of **method inheritance**, which is a radical departure from the mainstream of object-oriented programming languages. With **method inheritance** a method attached to a class is inherited by subclasses of the class: Messages sent to instances of these subclasses are handled by those inherited methods. With **method applicability** the classes of arguments supplied to a generic function determine which methods apply and are invoked. The Object System is derived by a major design effort from New Flavors and CommonLoops.

There has been virtually no experience with the Object System because there has only recently been available a real implementation of the Object System. All experience with CLOS-like systems has been with New Flavors and CommonLoops, which are not only different from the Object System but do not have metaobject systems like that which will most likely appear in the Object System.

I think, though, that the Object System represents one of the few advancements in object-oriented languages in the last 10 years, and CLOS-like languages should be at the forefront of object-oriented research, unless my fears are correct about research on standardized languages.

Here are what I think are the major contributions of the Common Lisp Object System:

1. Substitution of Method Applicability for Method Inheritance

Method inheritance works fine for unary operations, but when multiary operations on instances are required, the model breaks down in that the programmer often has to be concerned with the mechanism of multiple dispatches rather than with their interface.

People argue that method applicability is a generalization of method inheritance in that the "message" is the operation name, but this generalization gains only method applicability on the first argument. So-called multi-methods are a generalization based on a paradigm shift.

One nice customization technique that works well with generic functions is for a system to invoke multi-argument generic functions whenever certain important events or actions take place. By extending the class hierarchy with important subclasses and by providing methods on them, one is able to customize that system in an elegant way. The key factors are the extensibility of the hierarchy and the ability to describe a complex event by describing a combination of classes for the arguments. I believe there are further techniques based on this sort of model, but the key step has been taken by the Object System.

Delegation is the other competing object-oriented model. In delegation, prototypes are used instead of classes to implement shared behavior. The claimed benefits are that delegation and prototypes model how people learn and conceive of categories, that default values are handled more naturally, that some problems regarding the magic SmallTalk variable named `Self` are cleared up, and that interactive, incremental software development is easy. The first and last of these are simply claims whose validity is possibly more amenable to psychological and sociological methodologies than to reasoning. The problems with default values and `Self` seem to be the subject of low level language design.

Delegation seems to be weaker (but not in the formal sense) than the Object System because it mixes together different levels or areas of concern. With an inheritance system, the author of a particular piece of code or method can leave it up to the designers of the hierarchy and its other associated methods to best place methods in such places that other methods that need to delegate work can find them without knowing their names. That is, in message-passing and method-applicability systems there are means for invoking less specific inherited or applicable methods to complete the action. With delegation one can do this, but the author of the method needs to be aware of either the identity of the method or object to which responsibility is delegated, or the name of place where that identity is stored. This latter constraint is only a problem when there is a need to erect and maintain

very strict abstraction boundaries.

2. Inheritance from More than One Domain

Some people call this "multiple inheritance," but I think that that phrase carries too much baggage, at least in the form of objections based on a particular view or implementation of multiple inheritance. It seems plain that the behavior of objects should not be limited to those things inherited from one family chain: Some portions of a program may deal with an object using operations appropriate to what it represents to the user while other portions will use operations appropriate to how it is represented or how it is displayed on the screen. These two views cannot be reasonably captured by a single family chain; therefore the combination of chains is required. Multiple inheritance is a technique by which a class is created that represents a point in the combined family trees: Objects that have the combined behavior are instance of that class. Once the choice is made to inherit characteristics from several family chains, there are various decisions one can make about the behavior of the object when multiple ancestors supply same-named properties (slots and methods, for example). We will call such potential confusions **collisions**. One choice is to shadow, another is to signal an error, a third is to combine, and a fourth is to provide mechanisms for programs to selectively use one or the other of the possible properties.

The Common Lisp Object System does not treat all collisions uniformly. With methods, combination is used: with slots, inheritance, shadowing, and combination are used. In order to implement shadowing, a **class precedence list** is computed and single inheritance shadowing is used on this constructed family chain. A class precedence list is a total ordering of the classes with respect to a given class. This is not a very elegant solution, I think, but it is workable.

3. Method Combination for Method Collisions

Method combination is one of the most interesting concepts in the Object System. Had I total control over the design of the Object System I would have adopted some form of combination in all property collision situations. For example, I would have introduced the idea of domain or family into the language and provided a means for instances to be instances of more than one class simultaneously without having to provide a class to represent the combined object *as a combined object*. I call this **multiple, single inheritance**. With it one can do a plausible job of implementing a facility to view an object as an instance of some particular class, which is useful in database applications. A method then must specify the domains as well as the classes of its arguments. To completely solve all

inconveniences this approach introduces requires a more elaborate use of combination for methods in generic functions, but that is a topic for research and beyond these remarks.

Method combination is, as I say, one of the more interesting concepts in the Object System, and it has no close relatives in other languages. The idea that's interesting is that the effective method is not one that the programmer directly programmed, but one which is produced by composing code at runtime. The method combination language in the Object System is quite ugly, but it is one of the first serious attempts to do non-trivial program composition—I regard function composition using closures as trivial. I feel that such program composition is one of the two or three new computational models on the horizon.

4. First-class Treatment of All Aspects of the Language

I think the Object System doesn't go as far as it could in making everything first-class, but it goes a lot further than most any other object-oriented language. SmallTalk and C++ are examples of disappointing languages in this respect, though SmallTalk is quite advanced over C++. The metaobject protocol is possible because of this first-classness and is itself a major contribution. I think that the Object System does not have a rich enough structure to support the metaobject protocol in the manner it deserves, but it is quite good.

Oaklisp is probably the pinnacle of first-classness, though in its dumb way HyperTalk does pretty well, at least in exposing that which is normally hidden.

5. Initialization

Initialization, redefinition, and changing the class of an object are places where the Object System has moved ahead of other languages, but other weaknesses of the Object System have rendered these aspects unpleasant. The problem is that some arguments used for initialization fill slots and others supply keyword arguments to methods. This problem surfaces in two places: one is the interaction of these two roles for initialization arguments with the complexity of lambda-lists; the other is that the characteristics of the lambda-list that **make-instance** inherits come from various places in class-defining forms and from methods.

Both generic functions and methods use Common Lisp lambda-lists, which requires a complicated set of rules to determine argument defaulting and to define when the lambda-list of a generic function and the lambda-list of one of its methods are congruent. This

choice was made because some of the designers believed that method definitions should look almost exactly like function definitions. An initialization argument that is intended for some method or methods must make its way through a morass of complex lambda-list mechanism to get to the code that will use it. Had the interface between a generic function and its methods been made cleaner, not only would confusing syntax been eliminated but the entire initialization protocol could have been simplified.

The key idea of generic functions that I think can be pushed further is that a generic function is an interface to a very interesting piece of code—the combined methods. As an interface, generic functions could be made quite a bit richer, raised to a much higher level (certainly to a level that sits above a number of different implementation languages and their runtime systems), and possibly less focused on the performance required for functions that are called in inner loops. Such a model of generic functions would be useful in a parallel setting.

6. Conclusions

In short, the Common Lisp Object System sits well above the plain of ordinary object-oriented programming languages, and, if work continues in CLOS-like languages, it will be the wellspring of the interesting future object-oriented system research.

A Position Paper for the CLOS Workshop.
John Gateley, Texas Instruments.

Texas Instruments is currently researching programming languages and real time systems using Common Lisp. Optimization techniques for CLOS will be of importance in this project. TI will also be investigating the possibilities of using CLOS in a real time system. This will require determining, for example, whether any CLOS features require unbounded execution time, since these features could not be used in a hard real time system. A related issue is whether any features of CLOS compile efficiently on a lisp machine style architecture, but not on more standard hardware.

I feel this workshop is a great opportunity for me to learn more about CLOS and object oriented systems. In addition to the particular items above, I am also interested in more fundamental concepts, such as object oriented programming paradigms. My background is programming language design, and so I am also curious about design decisions made in the design process.

METHOD DELEGATION IN THE FORMS INTERFACE CONSTRUCTION KIT

Neil Goldman

USC\Information Sciences Institute

INTRODUCTION

The Forms Interface Construction Kit is intended to provide a body of software for building user interfaces to application programs coded in Common Lisp. In particular, the Forms Kit is concerned with interfaces built on a hardware base with a bitmap display with mouse and keyboard input devices, for applications running in an environment with a hierarchical window system. Support is provided both for the >>presentation<< of information in windows and for user and program >>interaction<< with the data through the input devices.
In designing the Forms Kit two goals were paramount:

- >>Portability<< -- The Common Lisp language specification does not provide constructs necessary for defining user interfaces. Forms Kit reduces portability problems by providing a relatively small "virtual workstation" which must be implemented for each Common Lisp environment. A wide variety of interfaces can then be implemented by composing the facilities of Common Lisp and this virtual workstation.

- >>Specification Oriented Definition<< -- The Forms Kit provides a compromise between constructing an interface by selecting from an interface library and programming an interface in terms of window system primitives. In essence, the Forms Kit provides a high level language for composing and modifying interface specifications. In contrast with interface toolkits in which interfaces are defined by example, Forms Kit provides the programmer with a full programming language for interface definition.

BASIC CONCEPTS

Forms Kit interfaces are built upon a hierarchical window system with rectangular windows. A form window may >>immediately contain<< one or more >>instantiated form<<s within it. Every instantiated form is either a >>hierarchy leaf<< or is hierarchically decomposed into other forms.

Each form has a single >>viewport<<. A viewport is an abstraction that serves two purposes. First, it provides the mapping between a form and the subregion of its window allocated for displaying its data. Second, since displaying the full form may require more space than is allocated to it, the viewport defines what subregion of the full form should be displayed at any given time. The ability to adjust the portion of a form that is visible is called >>scrollability<<.

The viewports for a window's top-level forms must occupy non-overlapping rectangular subregions of the window. The vieport of a form may be no larger than its parent's viewport, reduced by borders and margins. Sibling viewports may not overlap. Any non-leaf form appears as a tiled collection of subforms, surrounded by

border and margin. The scroll offsets determine which subforms are visible within a collection.

One of the major benefits provided by Forms Kit is the determination of actual viewport sizes and positions from abstract specifications in terms of features such as >>filling<<, >>spacing<<, and >>justifying<<, and >>aligning<<.

OBJECT ORIENTED PROGRAMMING AND FORM DEFINITION

Form definition plays the same role in interface specification that proceduralabstraction plays in programming. Naming an interface specification makes iteasy to reuse. It also make recursive specifications possible. Parameterizing these named specifications makes them far more useful. The parameters must encompass not only the data to be displayed, but aspects of the presentation and user interaction as well.

Many people have observed that user interface specification (both appearanceand interaction) benefits greatly from an additional kind of abstraction provided by object oriented programming, which permits modular specificationsto be combined, augmented, and even modified to form new specifications without the need to make lexical copies of the component pieces. CLOS has been adopted as the substrate for the Forms Kit to provide this kind of modularity.

In Forms Kit, form classes are implemented as standard classes. Instantiated forms are implemented as standard instances. Most generic functions are composed of methods that discriminate based on only one operand (although the generic functions often have other operands). The discriminated operand is always an instantiated form, and, by convention, it is always made the first operand of the function. Forms Kit contains macros which simplify writing code that conforms to this convention. These functions are called >>form generic functions<<.

STRUCTURAL INHERITANCE

In building interfaces with Forms Kit, we have encountered numerous cases in which a form of "inheritance" was desirable based not (solely) on class specialization, but on the hierarchical structure of our forms. In particular:

- Certain aspects of appearance, such as FONT and TEXTURE, should be specifiable for entire collections of forms and inherited through the parentage hierarchy. Most forms do not have their own specification for FONT, but use the font specified for their structural parent.

- In some cases, it is easier to specify the behavior of forms in response to user gestures (e.g., mouse clicks) centrally for a collection of forms, rather than distributed across the members of the collection.

- Sometimes a piece of application data need only be stored once, in a slot of a collection, rather than copied to every component that needs it. The components should not be aware of where in the structural hierarchy the data actually resides -- they should simply access the data with a generic function that "inherits" it from whatever structural ancestor holds it.

IMPLEMENTATION

Forms Kit provides a second form of inheritance, >>structural inheritance<<, that is specific to the hierarchical nature of its forms. Any form generic function can be declared, via a simple macro, to be a >>propogating generic function<<:

DECLARE-PROPOGATING [function-name {args} . {body}]

In essence, the macro simply provides a method for <function-name> specialized to MINIMAL-FORM, which is the most general class of instantiable form. The method simply reinvokes the generic function on the form parent of the form on which the method was invoked, passing the remaining arguments unchanged. <args> and <body> come into play only if the calls propogate through a root form (one with no structural parent). In that case, they determine the behavior, the default being to signal a "no applicable method" error.

A DECLARE-PROPOGATING-SETF macro is provided to create propogating SETF functions.

DISCUSSION

It seems that the concept of an object designating a >>delegate<< for itself for certain generic functions would have fairly broad utility, and generalizes suitably to functions that are generic in more than a single operand. The implementation described above hides the concept of delegation from the method/slot lookup components of the object system. The execution cost is pays for this seems fairly large -- an extra method invocation, with all operands
passed, for every layer of delegation. Do the proposed CLOS "meta" protocols provide a more efficient mechanism? What declarations could be made to enhance performance of method/slot lookup for delegating methods? Does the CLOS meta object proposal include portable means for extending the language of declarations?

The DECLARE-PROPOGATING macro completely fails to handle one important aspect of method delegation. In some cases, such as centralized control of user interface gestures, we find the the method which ultimately handles the call needs to know for which form its form operand is acting as delegate. In our implementation, this is handled by adding an additional parameter to the generic function. Originating calls always supply the same form argument in two form positions; the second occurrence is simply passed along unchanged as control is delegated up the form hierarchy.

In all our uses of propogating generic functions, we have wanted class inheritance to take priority over structure inheritance. We do, however, have a case in which we need a form of method composition that combined information computed from class-inherited methods with information from structure-inherited methods.

Our experience shows that often, but by no means always, an entire class choose to delegate in a uniform way.

Finally, we note that the DECLARE-PROPOGATING macro discussed above could be implemented in PCL, it apparently could not be done with the advertised CLOS interface. The reason is related to the overly-stringent, in our opinion, congruence requirement for lambda lists of methods of a generic function. Our macro-defined methods receive and pass the parameters that are not being changed with an &rest arg, although some of them are required parameters. This would not be permitted in CLOS, nor is there any advertised way for the macro to determine a generic function's "congruence signature" so that it could compute an appropriate lambda list.

Hiroshi Hayata
Fuji Xerox Co. Ltd.
System Technology Center

I have few experience writing CLOS code. I am interesting in application parts which is written in CLOS.

XAIE have many and various application parts, FDEV mechanisum, Programmer Assistant mechanisum, DWIM mechanisum, Windows, Tedit, Sketch, Grapher and so on, but they don't have independency as application parts. These parts run only on XAIE. So application users must use whole XAIE system to built application program with these parts. I think that these parts should be written in CLOS to get generic application parts.

I think that there are 2 ways to write these application parts. The first way is that specifaction of application parts are decided, class strucure is designed, and try to implement. The second way is that existing lisp program is used. Existing lisp prgram is incrementally rewritten in CLOS. The first way will takes long time, but can get well structured parts. The second way will not take so long time, but can not get well structured parts. Especially, I am not so familiar with CLOS. The second way seem to attractive to bignners of CLOS.

I want to discuss about techniques for converting to CLOS and programming methodology of CLOS.

IBUKI
Position Paper for CLOS Workshop

IBUKI is the vendor of IBUKI Common Lisp (IBCL). IBUKI has a major commitment to support CLOS as the objects system standard for Common Lisp. IBUKI's primary interest in attending the CLOS workshop is in determining what kinds of modifications to the kernel of IBCL will best optimize CLOS performance. IBUKI is committed to supporting such kernel changes so long as they are consistent with IBUKI's general commitment to providing small, efficient, and portable Common Lisp environments.

A Paper for AAAI CLOS workshop, prepared at 88.09.09, to be presented at '88.10.03 or 04

Masayuki Ida
CSRL ISRC
Aoyama Gakuin University
Shibuya, Tokyo JAPAN 150
ida%cc.aoyama@relay.cs.net

A-DISPATCH Algorithm for CLOS Method Look Up and its Application to CAM-based Accelerator

1. What is A-Dispatch algorithm

A-Dispatch (Associative-Dispatch) algorithm is an algorithm for CLOS method look up, which is intended to be suitable for our accelerator hardware pro- ject, but is generally applicable. The evaluation of A-Dispatch is not fin- ished yet. Though the early version of it was implemented on our prototype board, there are lots of rooms to fix and improve. This paper is not a com- plete paper.

2. Basic Idea behind A-Dispatch

Associator like (key1, key2, ... , keyn, value) is the issue of this paper. The simplest but not enough understanding is to store associators in CAM (content addressable memory) and retrieve the value for (key1, key2, ... , keyn) with one-cycle instruction. If the application is simple enough that the number of keys is fixed completely and the equivalence matching is the only required retrieval, it is very easy to implement. We are thinking about the acceleration of method look up which is not so simple. CLOS has a following characteristics to be considered for method look up im- provement.

1) The number of arguments which play roles in method discrimination is fixed for each generic function.

2) There is an inheritance which needs more than equivalence matching. Furthermore, the inheritance is not a single one but a multiple.

3) There is a dynamic behavior of determination. We cannot statically fix the relation among methods, and among classes.

4) The order in a class precedence list of each class is fixed at least just before the evaluation of it.

5) CLOS has a multi-method feature.

3. Key is class precedence resolution

If CLOS has a single inheritance nature, the situation is better for class relation determination. Class relation can be linearized and we can be easy to provide a mechanism to determine the relation among classes. Since CLOS has a multiple inheritance feature, relations among classes depend on the specification written in each classes. We need a class precedence list logically to trace super classes of a class. And this trace should be done each time we need the relation.

ANSI 88-002R Page 1-20 contains an interesting case;
(defclass pie (apple cinnamon) ())
(defclass pastry (cinnamon apple) ())
(defclass apple ()())
(defclass cinnamon ()())

The class precedence list for pie is (pie apple cinnamon standard-object t). For pastry is (pastry cinnamon apple standard-object t). Though it is not possible to build a new class that has both pie and pastry as superclasses, the above pie and pastry definitions have no problem.

Suppose we have the following defmethods.
(defmethod foo ((x apple)) ...) method1
(defmethod foo ((x cinnamon)) ...) method2

When (foo pie-instance) is executed, method1 must be dispatched. When (foo pastry-instance) is executed, method2 must be dispatched.

4. CAM structure.

We assume we have a CAM circuitry with the following specification.
enough bit length in word to hold several keys and a value.
enough number of words to hold associators.
fast interface between CPU and the CAM.

We assume the CAM has following operation menu.
multiple field masking for keys
multiple candidate detection
complete matching
arithmetic detection
GC-free store

The current technology of CAM satisfies most. (but not complete yet)

5. A-Dispatch algorithm

5.1. Assumption:
Associators for MCE (Method Cache Entry) and CPE (Class Precedence Entry) are pre-stored all. It is possible to modify/delete an existing associator and to create a new associator.

MCE = (generic-function-object attribute class-arg1 ... class-argN body)
N should be fixed for an implementation, say N=5.
(generic-function which has more than N arguments is out of scope)

CPE = (class attribute super-class1 ... super-classM)
M should be fixed for an implementation, say M=6.
(If a class has more than M super-classes, it is possible to expand using multiple candidate detection feature)

5.2. Algorithm:
This algorithm starts upon the final phase of invocation, that is, all the arguments are evaluated, and the generic function object is detected.

1) for each argument of call, get the class of it. Retrieve CPE and register ((The-class 0)(super-class1 1)(super-class2 2) ..) to the simple cache in the CAM board.

2) retrieve MCE and get the body.

A case for (foo pie-instance):
1) ((pie 0)(apple 1)(cinnamon 2)) is registered to the simple cache.
2) Since CAM contains (foo attribute apple method1) and (foo attribute cinnamon) are stored, (foo pie) has no entry with direct equivalence.
2-1) Retrieve MCE and get flags indicating there are several candidates.
2-2) Read out all the candidate associators.
2-3) Translate class-of-arg to the index consulting the simple cache.
(foo attribute 1 method1) and (foo attribute 2 method2) are made.
2-4) Arithmetic compare to choose the one which has lease value.
(foo 0) find out (foo attribute 1 method1) is the method we want.

A case for (foo pastry-instance):
1) ((pastry 0)(cinnamon 1)(apple 2)) is registered.
2)
2-1) Retrieve MCE
2-2) Read out all the candidate associators.
2-3) Translate class-of-arg to the index consulting the simple cache.
(foo attribute 2 method1) and (foo attribute 1 method2) are made.
2-4) Arithmetic compare.
(foo 0) find out (foo attribute 1 method2) is the method we want.

Multi-method is treated with much the same manner except 2-3) is looped for all the arguments. 2-4) needs only one time even arguments are multiple.

5.3. Performance Estimation:
N clocks for N arguments 1)
+ 1 clock for retrieve MCE 2-1)
+ X minor-clocks for 2-2).
+ Y clocks for translation 2-3). Y can be 1.
+ 1 clock for 2-4)

---

3 + N + X'

If the clock of the board is 10Mhz, N=3, and X'= 3 (major-)clock, then the total time is estimated to (3+3+3) X 100ns = 900 ns. It is important that there is no firm evidence for the above performance data.

6. The early experiences of A-Dispatch

Since 1986 spring, we have been trying to make an experimental system. We made an experimental system using a 68020 VME Un*x sV system. I designed and made an accelerator board with 8 NTT CAM chips. The chip we used has the following data;

word organization: 128 X 32 bit (expandable to more bit width)
operation mode: 30
cycle time: 140 ns
chip size: 10.3mm X 8.4mm
total elements: 71,300
Process: 3um CMOS metal double layered
power consumption: 250mW (5Mhz)
pins: 53 (+ 9 for power)
manufacturer: NTT for experiment
(This chip is not showing the best technology NTT has.
their current chip is 512 X 40 bit (100 ns or less) with more
operation modes)

Since we designed 32 bit X 1Kw CAM array, classes and methods are stored by sequential index and N is limited to 2. Maximum number of classes is 256. Maximum number of generic functions is 1024. This board has two versions. One is a hand-made prototype made during the design phase of Fall 1986 to March 1987. The other is a wirelapped one with improved circuitry made dur- ing 1987. The cycle time of the experimental board is set to 4 Mhz which is slow enough to trace. The interface uses memory mapping technique. Now, we are in the third phase to design more realistic implementation of A-Dispatch algorithm.

CLOS Workshop Position Paper
September, 1988

Uses of CLOS in the Xerox Intelligent Information Access Project

We plan to develop user interface and information access technology in Common Lisp on a range of machines, including Mac II, Xerox Lisp machines, Sun workstations, and Silicon Graphics IRIS workstations. We would like to evolve a common framework for program development for:

- window manager interactions
- graphics: from 2D monochrome to 3D color animation
- network access
- database management, including a file-based B-tree package
- linguistic tools

Some of this software already exists (in Common Lisp, other dialects of Lisp, and C) and some has not yet been started; very little of the system integration has been completed. We plan to convert everything not already in Common Lisp to Common Lisp.

Much of the software we plan to build and integrate conforms readily to an object-oriented structure. Our hope is that using CLOS as an framework for all the pieces of our project will increase our productivity and decrease the costs of building and maintaining our complex software system. Our fear is that CLOS will extract a performance penalty that we will not be able to tolerate (we are very sensitive to performance, particularly in the user interface to this system); we have heard of people who have tried to use PCL but eventually abandoned it because of performance problems.

We hope to achieve two things at the workshop. First, we would like to find other people with software needs similar to our own. For example, if someone is building a machine-independent window system interface, we would like to collaborate with them. Second, we hope to find out what plans are being made for increasing reliability, stability and performance of CLOS.

Sara Bly
Stu Card
Doug Cutting
Baldo Faieta
Kris Halvorsen
Austin Henderson
Herb Jellinek

Walt Johnson
Jock Mackinlay
Jan Pedersen
George Robertson

Vaughan Johnson
Mike Hewett

I am developing a new object-oriented version of the BB1 Blackboard Control system in PCL. Knowledge in BB1 is represented in a semantic net, in a style partly similar to Sowa's Conceptual Graphs. We have a type hierarchy embedded in the net. Each link between objects has a dual in the opposite direction. The primitive types of objects we represent in the hierarchy are:

- generic objects (user-specializable),
- link types,
- knowledge sources,
- language features (including event, state, and action verbs; modifiers; nouns; and sentences),
- roles,
- collections (various groupings, including contexts and graphs),
- knowledge bases, and
- application systems ("skills" in our terminology). Multiple skills can be loaded and can interact.

What's interesting from our work with respect to CLOS is the way the CLOS object hierarchy fits/doesn't fit the BB1 semantic net representation. For example:

The closest conceptual mapping of the BB1 semantic net/type hierarchy to the CLOS class hierarchy would have every BB1 object be a CLOS class. However, we need specific attributes for BB1 objects, so we'd need to use a single instance of each of those classes, to get specific attributes as slots. The high overhead of defining classes prohibits this approach, because BB1 applications often have several hundred objects. The approach we have planned is to make certain special types of BB1 objects (specifically knowledge sources and their instances, link-types, skills, and graphs) be instances of separate CLOS classes, and all other BB1 objects be instances of a CLOS class for generic BB1 objects. Also, in BB1 we have three levels of hierarchy-membership: class, individual (a.k.a. example), and instance. This additional distinction, and the different semantics we have for class and instance, make the BB1 type hierarchy/semantic-net to CLOS hierarchy a problematic fit.

We want to have arbitrary (naming, semantics, arity) links defined between objects. The links themselves are not BB1 objects, but because we want to be able to reason about types of relations, we want each CLOS link-class for our links to have a corresponding BB1 link-type object in the type hierarchy. The CLOS link-class must be related to the BB1 link-type so that, for instance, we can reason about the type of a given link by finding out its CLOS class, and from that the position of the BB1 link-type in the type hierarchy. We often want to follow classes of links in searching the semantic net, e.g., the class of "is-a," "instantiates," and "exemplifies" links -- all definitional in the type hierarchy. Using the type hierarchy to define the classes is important to reasoning in BB1, so it is important to be able to quickly determine the BB1 link-type for a given link between BB1 objects, so we can use the position of the link-type to determine membership in the search class.

We want the trigger and precondition checks, and the actions, of BB1 Knowledge Sources to be methods, but also be easy to extract and edit, some even at run time.

We want to be able to save knowledge bases as modules, even if there are links between BB1 objects in different knowledge bases. Then, when loading these knowledge base modules, we need links between objects in different knowledge bases to be instantiated correctly.

I will bring some diagrams to the workshop, that show our CLOS classes and the way they implement BB1 objects.

We are, of course, concerned with the performance of CLOS. Two of the big reasons we embarked on an object-oriented implementation are reduced code size (through code re-use) and increased efficiency (because BB1 objects are modeled well by object-oriented notions). It appears that PCL meets our needs quite well in this regard.

Position Paper for CLOS workshop October 3-4, 1988
By Jeremy Jones and Gary Byers of Coral Software

Coral is going to begin work on a native implementation of CLOS for Coral Common Lisp (CCL) on the Macintosh (CCL is also known as Allegro Common Lisp for the Macintosh). In addition we will be re-implementing our window system in terms of CLOS. This paper describes some of the issues we expect to face and asks many of the questions that we will need to answer in the course of our implementation.

Coral Common Lisp currently takes up less than 800k on disk including our window system, editor, and all of our development tools. The current native object system is Object Lisp which takes up 4-5k. We have found that PCL adds from 300k to 400k to the size of our Common Lisp. Thus converting from Object Lisp to PCL represents a fifty percent increase in the size of CCL! This fact alone necessitates a native implementation of CLOS.

In order to proceed with our native implementation we would first like to understand the implementation of PCL. We would like an overview of PCL. What are the important data structures that are used? Which functions are the heart of the system? What are the major bottlenecks and sources of inefficiency? What things were tricky or difficult to implement correctly? What takes up all of the space? What types of things would have been done differently if portability was not a consideration? What are the time/space/safety tradeoffs?

The next consideration is how we make an orderly transition from PCL to a native implementation without writing CLOS from scratch. Which parts of PCL should we keep (if any) and which parts should be thrown away? Since PCL has its own code walker it seems that we should get rid of the code walking in PCL and have it done by our compiler. What is code walking used for besides implementing `symbol-macrolet`? Are there any other cases of things which are likely to be redundant with internal system capabilities? Bootstrapping is obviously a difficulty in PCL. What kinds of native support would be needed to simplify or eliminate the bootstrapping procedure? What portions of PCL are used only for bootstrapping and can thus be eliminated from a delivered product? Are these portions currently reclaimed by a garbage collection?

How do make our CLOS implementation as small as possible? Assuming CLOS is inherently big (it has a lot of functionality) can we layer it so that only a reasonably small CLOS kernel needs to be included in our kernel and the rest can be optionally loaded in? This is important since our minimum memory configuration is only one megabyte and not all users will want to use CLOS. It seems fairly straight forward to implement the commonly used functionality of `standard-class` but how much of the meta-object protocol needs to be there?

How do we compile down applications based on CLOS so that only the minimum set of components of CLOS and only the necessary classes and methods will be included? This is an important consideration since the biggest weak point of Lisp is delivery of applications. There should be standard techniques for producing small stand-alone applications written in Lisp and CLOS. What kinds of declarations and proclamations need to be added? Could some classes be proclaimed as unused so that none of their methods would be included in the final application? Should we add declarations to say that a class or generic function is finished being developed and won't be specialized or redefined so that the compiler can make assumptions about the run time? Is it possible to use the garbage collector to remove classes and methods that are not used in an application? Is it possible to use type declarations at compile time to determine which classes and methods

need to be there at run time? What types of things need to be considered when designing a system in CLOS that is intended to be compiled to an application?

Coral currently has a window system based on Object Lisp, we are redesigning our window system to be based on CLOS. The first problem we are faced with is that many of the unusual dynamic features of Object Lisp are used, such as adding and removing instance variables on the fly, instantiating classes, or creating sub-instances. Once these are cleaned up and put into a traditional class/instance format, a number of stylistic issues still remain.

The easiest way to translate Object Lisp into CLOS is to turn all `defobfuns` into primary methods and all `usuals` into `call-next-method`. Is this an appropriate way to design a programmer interface to a window system? What are the performance penalties of `call-next-method` in relation to before and after methods? Is `call-next-method` considered an advanced feature with before and after methods considered more appropriate for beginners or non-sophisticated users? We would like to base our window system on a subset of CLOS so that all of CLOS need not be loaded in order to use our window system. What subset should we use? Should `close` be turned into a generic function that operates on both streams and windows, or should we use `window:close` or `window-close`? Should `setf` be used to set characteristics of windows that have graphic side-effects such as position and color?

Does anyone have any good ideas on CLOS browsers and debuggers?

A Design for High Performance Dynamic Generic Dispatch in the Common Lisp Object System

James Kempf
Sun Microsystems
2550 Garcia Ave.
Mountain View, CA, 94043
415-336-4601
jkempf@sun.com

Keywords: generic dispatch, method lookup, Common Lisp Object System, CLOS, method invocation performance

Abstract

This paper examines an algorithm for generic dispatch which combines open addressed hashing for direct method lookup with a ternary tree for inherited method lookup. A classical optimization procedure is used to determine an optimum size for a hash table containing direct methods. A cost function is formed which weighs the number of probes required to retrieve an item as a function of the table size against the amount of memory needed for storing the table, and the cost function is minimized with respect to the table size, yielding an optimum table size. Similar procedures, though common throughout a number of engineering disciplines, are rarely formally used in software engineering. To speed inherited method lookup, a scheme for assigning classes unique identifiers based on their class precedence lists is described, and data from three inheritance hierarchies in the literature is presented to support the claim that the encoding scheme is not impractical. Finally, a ternary tree based algorithm for inherited method lookup is described. A design for a method lookup algorithm is suggested which uses a hash table when the number of inherited parameter specializer combinations is small, and a ternary tree with a hash table cache when the number of combinations is large.

Introduction

The design of the Common Lisp Object System (CLOS) [ANSI88] as part of the ANSI X3J13 standardization of Common Lisp has provided Common Lisp with a standardized object-oriented extension that is taylored for application programmers

interested in developing portable application software. An important feature of CLOS is generic functions. Generic functions are functions with a localized interface and a distributed implementation [Moon86], in contrast to regular functions which have a localized interface and a localized implementation. As in other object-oriented languages, user-defined methods provide the implementations for generic functions, but, unlike other object-oriented languages, methods in CLOS can be defined to use any number of parameters for generic dispatch. Most object-oriented languages restrict generic dispatch to the first parameter, and often even the first parameter is hidden. Generic functions are nothing new for Lisp, since many built-in functions (such as +) are already generic, but system support for user-defined generic functions is new.

In order to make generic functions attractive for application developers, generic dispatch must not impose a significant penalty on function invocation. Previous studies [Deutsch83] [Krasner83] [Cox86] have found that a global hash table based cache can improve method lookup performance considerably, but whether these results extend to generic functions and multiple parameter dispatch is not clear. In this paper, theoretical results for open addressed hashing are used to form a cost function giving the cost of a hash table as a function of the amount of memory utilized, and the cost function is minimized, to yield an optimum hash table size for a given number of methods. Optimization techniques, such as those applied here, have seen wide use throughout other branches of engineering, but, for some reason, most software engineering design studies have failed to utilize them, even though theoretical results on the performance of important algorithms have been available for many years. The results of this optimization procedure are not specific to generic dispatch, but applicable wherever open addressed hashing is a part of the design.

The hashing algorithm addresses direct method lookup and lookup of inherited methods when the number of combinations of subclasses in the specialized parameter list is small. When the number of parameter specializer combinations is large, however, a different algorithm may be needed because the amount of memory required for the hash table becomes unwieldy. To better support inherited method lookup, a novel encoding scheme is suggested for assigning unique ids to classes. The scheme derives a class's identifier for use as a hash and search key by encoding the class precedence list. Evidence is presented from three inheritance hierarchies in the literature [Goldberg86] [ANSI88] [O'Brien87] that the encoding scheme is reasonable for real systems. A ternary tree based algorithm is described for inherited method lookup. The results suggest a hash table for generic functions where the number of inherited parameter specializer combinations is small, and a ternary tree with a hash table cache when the number of combinations is large. The dispatching algorithm assumes one dispatch table per generic function, as in the Portable Common Loops (PCL) implementation of CLOS [Bobrow86].

The next section surveys previous work in enhancing the performance of generic dispatch for other object-oriented languages. In Section 3, the optimum size of the hash table is derived, as a function of the number of methods defined on the generic function. Section 4 develops the encoding scheme for generating unique class ids, based on the class precedence lists, while Section 5 presents the ternary tree based algorithm for inherited method lookup. Finally, Section 6 summarizes the paper.

Previous Work

For single inheritance, statically typed object-oriented languages (such as C++ [Stroustrup86]), inherited methods can be efficiently implemented by a pointer to

the method function at a fixed offset in the object. When a method invocation is compiled, code for retrieving the function pointer through an index into the object's inherited method function table, and for invoking the method function indirectly via the function pointer is generated. Such an implementation is extremely efficient, since the overhead is merely the cost of a memory reference and an indirect function call substituted in-line at the site of the method invocation. Because the classes cannot change during execution, the inherited method table need not even be kept in a class object. For methods which are not inherited, the overhead of maintaining a pointer to the function in the object can be dispensed with entirely, and the method invocation can simply be compiled as a direct function call in line, since the type of the first parameter is known at compile time and the address of the function is known at link time.

In single inheritance, dynamically typed languages, such as Smalltalk [Goldberg83] and Objective-C [Cox86], a program-wide cache as well as indexing into a class-based method function table have been used to speed method lookup, especially for inherited methods. The usual technique is to maintain a program-wide cache of most recently used method functions, with the class of the first parameter and the method name (or selector) as hash keys [Cox86] [Krasner83]. In [Cox86] and [Cox84], a program-wide cache improved method invocation time to an average of about 2.5 times a function call from a worst case of 70 times when lookup of an inherited method was required. Indexed lookup had an invocation time of about 2.0 times a function call, but the indexed implementation required considerably more space (one machine word for each method in each class). A figure of 95% is cited for the cache hit ratio (ratio of cache hits to total method invocations) in typical applications.

バイトコードによるSmalltalkでも同様のキャッシュによる効果が報告されている。

Similar figures have been found for caching in Smalltalk in an implementation based on byte code interpretation [Krasner83]. Overall system performance was found to improve by about 20-30% with a cache hit ratio of 85-99%. The execution speed was judged to be only between 1.0% and 4.5% slower than the optimal method invocation time assuming zero method access overhead. Since Smalltalk is fully object-oriented, comparisons with function calls are not possible. In [Deutsch83], a further refinement is described. In addition to the global method cache, an in-line cache of the most recently used method is maintained at the method invocation site. The invoked method's address and the address of the class object are dynamically linked in at the invocation site if a cache miss occurs, and a hit the next time through the method results in no overhead except for verifying the in-line cache's validity. Together with an optimizing translator to machine code, an implementation performed about twice as fast as the byte code interpreter. Note that the actual overhead of an in-line cache in the case of a cache hit is only one instruction more than for a singly inherited, statically typed language, and that instruction is the check for the validity of the cache. The average overhead will be slightly more, and will depend on the hit ratio, which, in turn, will be a function of the application mix.

In-line caching involves self-modifying code, since the results of a lookup when a cache miss occurs must be inserted into the machine instruction for the function call. Many Lisp implementations on conventional architectures have shared code segments for multiple users, a result of their evolution on time sharing machines, and these shared code segments are not writable. In addition, Lisp implementations often have relocating garbage collectors, which can cause the addresses of method functions and class objects to change. As a result, in-line caching is sometimes not possible for these Lisp implementations. However, the Portable Common-

S O L O :
A Portable CLOS Window Interface

Hans Muller, John R. Rose, and H. Tayloe Stansbury
Sun Microsystems

Soloは，X，NeWS，SunViewへのポータブルな，CLOSに基づくインタフェースである．このインタフェースの本来の目的は，まず，アプリケーション・プログラムが，下部に存在するウィンドウ・システムの違いに依存しないようにする，小型でオーバ・ヘッドの少ないソフトウェア層を提供することである．そして，二番目は，OPEN LOOK TM の仕様に基づいたユーザ・インタフェースtoolkit をサポートすることである．Soloは，その他のウィンドウ・システムにも，対応できるように，また，他のユーザ・インタフェースtoolkitsをサポートするように設計されている．

Soloは，X やNeWSで提供されているものに相当するいくつかの機能を持っている．それらは，入出力，入力用透明キャンバス，イメージ，フォント，カラー，出力プリミティブ群，インタレスト／多重入力プロセスをサポートするNeWSのような入力システムである．

我々の設計は，Interlisp Window System やCommon Windowsのような，いくつかの領域で現在用いられているLispのウィンドウ・システムから出発している．最も重要なことは，メソッド結合，キャンバス，キャンバスの幾何，イベント処理，見た目／感触である．

メソッド結合／拡張性

Soloは，Common Lisp Object System(CLOS) で実装されている．スロットやメソッド継承のために特別な機構を作るよりもむしろ，Soloは，そのデータ構造や操作を定義するためにCLOSを用いている．Soloによって提供される基本的な型は，CLOSのサブクラスや総称関数によって，カスタマイズや拡張が行われるようにしている．

キャンバス

Soloは，３種類の入出力の画面を提供している．：不透明キャンバス，透明キャンバスとビューポート・キャンバスである．最初の２つは，相当するNeWSの型と同等である．もうひとつは，スクロールをサポートする不透明キャンバスのサブクラスである．キャンバスは，できる限り基本的なものにしている．例えば，それらは，境界もタイトルも持たない．そのような特徴をもつように，適当にキャンバスを特殊化することは，ユーザ・インタフェースtoolkitに任されている．

キャンバス幾何

キャンバスの入出力の位置は，固定座標系ではなく，キャンバス固有の座標系に相対で指定される．これは，（キャンバスの端に相対の）原点の座標や，x y 座標が増加する物理的な方向を仮定するようなプログラムを書いたり，移植したりするのに，特に有効である．キャンバスの寸法は，比較的単純に指定される．たとえば，ビューポート（３つのキャンバスの型の中で，最も複雑）の幾何は，３つのアクセス関数で指定される．（最初の２つは，他のキャンバスの型にも適用される）：

(extent-region canvas)
ビューポートが表す論理的な入出力画面の範囲を表す．特に，ビューポート・キャンバスでは，その領域範囲のある部分のみを表示する．全ての入出力の位置は，その領域範囲相対に指定される．

(bounding-region canvas)
親キャンバスの領域範囲に相対なキャンバス全体を包含する領域を返す．
setfすると，キャンバスの形の変更／移動が起こる．

(view-port-region viewport)
キャンバスの中で可視な領域範囲の部分を返す．
setfすると，ウィンドウのスクロール／形の変更が起こる．

イベント処理

キャンバスは，Lispのlightweghtイベント処理プロセスと関連している．キャンバスは，ウィンドウ・システムがイベント処理プロセスに引渡してほしいようなイベントにインタレストを表現しなければならない．マウス・イベントは，マウス・カーソルの下にある最も上のキャンバスと関連したイベント処理プロセスに渡される．キーボード・イベントは，キーボード・フォーカスを持ったキャンバスと関連したイベント処理プロセスに渡される．また，損傷や他のイベントのクラスは，影響を受けたキャンバスと関連したイベント処理プロセスに渡される．マルチプル・ディスパッチが，特別なキャンバスの型やイベントの型に応じて，イベント・メソッドを付け加えるために用いられている．キャンバスは，"mouse-input-filter"スロットの指定によって，マウスやキーボードのイベントを，それが覆っているキャンバスに対して渡すことができる．

見た目／感触

OPEN LOOK のユーザ・インタフェースtoolkit は，基本的なキャンバスの型を特殊化して，視覚的に気に入った境界，タイトル付きのウィンドウ，メニュー，ボタン，スライダなどを提供している．OPEN LOOK ユーザ・インタフェースは，UNIX TM へのインタフェースとして，AT&Tのために，Sun によって開発された．仕様書は，Sun を通して入手可能である．（Contact kal@sun.com or dayyee@sun.com.）

我々は，CLOSのXerox PCL 実装上で，Soloのプロトタイプ作成を終了した．そのプロトタイプは，SunView 上で動作するLucid Window Toolkitを目標とした．プロトタイプを用いて，部分的なOPEN LOOK のユーザ・インタフェースtoolkit や，いくつかのデモ用のアプリケーションを作成した．このプロトタイプの経験をもとに，設計の改良を加え，製品レベルの実装を予定している．

我々は，オブジェクト指向ウィンドウ・インタフェースに興味のあるLisp関係の他のメンバーと共に仕事がしたいと思っている．もし，Soloの仕様のレビューに興味があれば，humuller@sun.comに連絡をください．

MICE: A Modular Intelligent Constraint Engine

Harley Davis

Sanjay Mittal

(Xerox PARC Systems Sciences Laboratory)

はじめに

MICE は制約充足問題を定義してそれを解くためのツールであり、モジュラーな知的探索エンジンを用いている。複数の探索技術を仕様として与えるとそれから特定の問題解決アルゴリズムを生成する枠組みを利用しているという意味で、探索エンジンは、モジュラーである。それに加えて MICE は、制約や変数、定義域の多様な表現をサポートしている。MICE は Xerox CommonLisp と PCL とで書かれている。ユーザインタフェースは Xerox ウィンドウシステムを利用している。

その詳細

制約充足問題(constraint satisfaction problem, CSP)は、定義域を持つ変数の集合とそれらの変数につけられた制約の集合とからなる。シリアルな計算機上では、CSP の問題解決機は一般にバックトラックアルゴリズムの形式をとる。CSP に関する研究は特定の問題解決戦略に焦点を置く傾向にあり、特定の定義域、特定の制約表現、特定の型の問題をよく仮定する。MICE では、多数の問題オブジェクトのクラスをそれぞれ異なった表現を用いて定義して、それらの間でを最小のプロトコルを共有することにより、複数の問題解決戦略を一般的なバックトラックの枠組みの中に統合することができる。

定義域や制約といった各問題オブジェクトはクラスの集合の中から選ばれる。クラス階層グラフのルートノードにおいてプロトコルは定義され、そのプロトコルはすべてのオブジェクト表現に適用可能であり、どんな問題解決戦略にも用いることができる。より特殊な表現は、より特殊なプロトコルを定義して、一部制限された戦略を用いるであろう。現在の版では問題解決戦略はメソッドとして表現され、その中で用いられる問題オブジェクト表現の個々の組合せに依存した形式となっている。CLOS のマルチメソッドがこの機能について特に有用である。

われわれはさらに、解決仕様のオブジェクトを定義した。解決仕様オブジェクトは多様なスロットを持っており、それぞれのスロットは全体の問題解決戦略の異なる部分を示している。このとき一般的な問題解決アルゴリズムは解決仕様オブジェクトの一部によって値を埋められる。メソッドの結合がより効率のよいアルゴリズムに還元できるときには、解決仕様のクラスはより特殊な問題解決アルゴリズムを包含する場合もある。たとえば、単純な時間順序に基づくバックトラックで要求される状態の情報は、他の知的なメカニズムに比べて少なくてすむ。このとき時間順序による解決仕様はより単純な枠組みを使うことができる。

こういったことから、オブジェクトのグラフは特別なものとなる。解決仕様オブジェクトの各スロットには多数の選択肢があり、そのほとんどは他のスロットの選択と組合せて動かすことができる。こういったことを表現する一つの方法は、各スロットの各選択にミクシン(mixin)クラスを用意して、これらミクシンのすべての組合せに基づいて問題解決メソッドを定義することである。しかしこの方法によると、クラスとメソッドの数が爆発的に増えてしまい(しかもその多くはとてもよく似ている)、とても保守できるようなものにはならない。これとはまた別の方法として、非常にジェネリックな問題解決戦略を用意して問題解決のサブクラスは保守しない方法がある。これはユーザを信頼して、問題解決機を走らせる前に解決仕様オブジェクトのすべてのスロットを埋めておいてもらうものである。このやり方だと、解決仕様オブジェクトのスロットの組合せにおける自然な協調を実現することができない。

われわれが選択したやり方はこれら2つの方法の中間に位置するものである。完全にジェネリックな問題解決機を定義して、最もジェネリックな解決仕様オブジェクトに少数のサブクラスを用意する。それぞれのサブクラスにはより特殊な問題解決メソッドが定義されている。解決仕様オブジェクトのどのスロットもユーザによって変更することができるが、より特殊な問題解決機は組合せて使うことができない。このようにミクシンオブジェクトの大きなラティスとその組合せ(特殊なメソッドを多く含む)、一般的だが遅いインタプリタを持つオブジェクトの小さなグラフとの間に競合が生じるが、これはオブジェクト指向プログラミングではよく起こることである。適切な解決は特定の領域に大きく依存しており、一般的な解決はむずかしい。

異なる問題解決モジュール間にはしばしばかなりの相互作用がある。その中には問題状態の生産者消費者というようにお互いに補いあうものも含まれる。このようにモジュール間の通信のために使われるような探索状態オブジェクトが定義されている。探索状態オブジェクトは特殊化するためにサブクラスを持つことができ、問題解決モジュールと密接に作用する。この解決策は理想的なものではない。生産される状態も消費される状態も一つのパッケージの中に収められた、より大きな粒度の問題解決モジュールを定義することによって、画一的な状態オブジェクトに頼らなくても済むようにするのが、究極の目標である。このようにすれば、使われないモジュールは状態オブジェクトを定義する必要がなくなるのである。

MICE インタフェースにおけるクラスグラフは問題解決オブジェクトのクラスグラフとほとんど相似形である。一つのインタフェースユニットは問題定義ウィンドウと、問題解決仕様ウィンドウ(といくつかの制御メニュー)とからなる。問題を定義するウィンドウは問題の各定義域、変数、制約用のセルを持ち、各オブジェクトについての情報を表示する。セルは問題オブジェクトエディタの中に展開することができる。エディタのフォーマットは問題オブジェクトのクラスに依存して決まる。定義域エディタは制約エディタとは異なる。拡張した定義域のエディタは値域のエディタとは異なる。問題解決ウィンドウでは次の問題解決機の実行の際に用いる戦略の集合を選択することができる。表示されたオブジェクトはすべて、それが指示する問題オブジェクトとつながるが、問題オブジェクトを妨害することはない。このようにして、複数の MICE インタフェースを開発することができる。

CLOS の問題点

探索問題では実行速度が最重要問題である。モジュールや問題解決の枠組み全体の最適化は、PCL にかなりの計算負荷を課した。PCL の実行速度は確かに向上しているが、結局は定数倍の速度向上に止どまるであろう。理論的には面白くないかもしれないが、アプリケーション分野において現実的にシステムを使おうとすると非常に高い実行速度が要求される。したがって、実行速度の問題を強調しておきたい。

メタオブジェクトプロトコルもまた面白い。われわれは問題のクラスを定義して、問題のサブクラスは問題オブジェクト(制約、変数、定義域)を継承するようにした。現在この継承を複雑な初期化手続きを用いてサポートしなくてはならないが、問題のメタオブジェクトを定義することによってさらに改良されるであろう。

CLOS の仕様の一部の側面は、実際問題では欲求不満を起こすものである。たとえば、オプショナル引数を特定化するのはしばしば有用である。これは適合ラムダリストの概念を損うことなく CLOS に統合されるだろう。

最後に、MICE をプログラムしている際に PCL の統合的な環境がないのがしばしば不満であった。この問題は CLOS の実際の仕様とは離れたものであるが、このワークショップが CLOS プログラマが必要としているツールについて討論するフォーラムになるであろう。

付録

1. MICE クラスグラフの一部

2. MICE インタフェースのスナップショット

a. 解決仕様メニュー
b. 定義域、変数、制約セルを持つ問題記述メニュー

CLOS Workshop まとめ
The Importance of Being Meta
D.G. Bobrow and G. Kiczales (Xerox PARC)

横尾 真 (NTT 情報通信処理研究所)

概要

Lispの提供するもっとも重要な機能は、問題に対して特殊化された言語をLispを用いて構築できることである。その言語はLispの拡張であり得るし、Lispの中に組み込むことも可能である。 Lispと特殊化された言語の両方を用いてプログラムを書くことができる。その様にして記述されたコードは分かりやすくかつ効率的でもある。

2つの主要なLispオブジェクト指向言語（FlavorsとLoops）は、このLispが持つ機能を持っていない。標準的な実行ルールを特殊化したものに従うようなオブジェクト指向言語のプログラムの要素を定義するメカニズムはサポートされていない。例えば、プログラマはインスタンス変数へのアクセスのルールが特殊化されたようなクラスを定義して、そのクラスを他の標準的なクラスと共に用いることはできない。

CLOSはCommon Lispの拡張であり、Common Lisp自身をオブジェクト指向言語にするものである。型とクラスの統合はすべてのLispのデータ構造がオブジェクトとして扱えることを意味し、関数呼び出しとメッセージ送信の統合はジェネリックなディスパッチが呼び出し側にとってトランスペアレントであることを意味する。メタオブジェクトプロトコルはLispの持つ機能がオブジェクト指向言語でも利用可能となることを保証する。

1 メタオブジェクトプロトコルはLispの持つ機能を提供する

文書化されたメタオブジェクトプロトコルの存在により、CLOSはLispの持つ特殊化された言語を構築して組み込む機能を保持している。 Lispではこの機能はプログラムの構造を直接操作できること、マクロを用いて言語を拡張できることによって実現されていた。

メタオブジェクトプロトコルは、基本的なプログラムの要素はメタオブジェクトと呼ばれるクラスオブジェクトであることを規定している。 CLOSでは、ユーザプログラムはメタオブジェクトを直接操作することができる。これはプログラムの構造を操作できることに相当する。また、メタオブジェクトのクラスを特殊化することができ、これはマクロを用いて言語を拡張することに相当する。

2 標準化の危険性

標準化が意味があるかどうかという疑問は、以下の3つの疑問に分割することができる。

2.1 CLOS は有用か?

CLOS の基本的な機能は、Flavors,Loops,CommonLoops 等の機能を改定したものであり、改定は慎重に議論を重ねて決定されている。メタオブジェクトプロトコルは CommonLoops にしか存在しなかったが、すでに多くの大きなプロジェクトにおいて用いられている。

2.2 CLOS は効率的にインプリメント可能か?

すでに CLOS を効率的にインプリメントするためのテクニックは存在するし、メタオブジェクトプロトコルの存在が効率的なインプリメントの障害にならないことも、PCL のインプリメントを通じて示されている。重要なことは、CLOS のパフォーマンスは正しく測定されなければならないということである。総称関数呼び出しに必要な時間は、通常の関数呼び出しに要する時間の 2 倍程度であるが、このことは CLOS のプログラムのパフォーマンスが Lisp プログラムの半分であることを意味するものではない。総称関数呼び出しに必要な時間は、通常の関数呼び出しに必要な時間のみではなく、型によるディスパッチを行い、適切なコードを見つけるのに要する時間とも比較されなければならない。[1]

2.3 標準化は研究をつぶすか?

この問題が最も重要であるが、現実はその逆であるように思われる。メタオブジェクトプロトコルはプログラムの隠された構造を明らかにしており、プログラマが容易に様々な構造に関する実験をすることを可能にしている。 CLOS は研究をつぶすというより、実験を行うための道具を提供することにより研究を加速している。 PCL のコミュニティでは、Loops や Flavors よりも多くの実験が行われている。メタオブジェクトプロトコルは現在の仕事の方法を提供するというより、共通の基盤の上に将来を探索する方法を与えている。

[1]bit 別冊 Commpn Lisp オブジェクトシステム p.21 の議論とほぼ同じ

ＣＬＯＳにおける高速なメソッドディスパッチ　　(解説：大久保　清貴)

A Design for High Performance Dynamic Generic
Dispatch in the Common Lisp Object System

by James Kempf

オブジェクトシステムの性能を決定するのは、
① インスタンスの生成に要する時間
② スロットアクセスに要する時間
③ メソッドディスパッチに要する時間
である。この論文では、メソッドディスパッチを高速に行なう為のテクニックについて述べている。

まず既存のテクニックとして、①インデックス法、②グローバルキャッシュ、③クラス毎にキャッシュを持たせる方法、④インラインキャシュを紹介している。①インデックス法は、全メッセージにユニークな数値を割付け、各クラスに大きな配列を持たせ、レシーバが受け取る可能性のあるすべてのメッセージについてその数値をインデックスとする要素にメソッドの場所を格納しておく方法である。この方法では配列の値を持ってくるだけで起動すべきメソッドが求まるので非常に高速であるが、非常に大きな配列がクラス毎に必要なので非現実的である。②グローバルキャッシュ法は、レシーバのクラスとレシーバの名前をキーとしメソッドの場所を値とするキャッシュをシステム内に１つだけとる方法である。メソッドを毎回サーチするのに比べ、一旦捜し出したメソッドをキャッシュに登録しておけば大幅に速度が改善される。③クラス毎にキャッシュを持たせる方法は、②の場合よりキーの衝突を減らす事ができ、さらに高速化が期待できる。④インラインキャッシュ法は、全てのメソッドの呼出し側に前回どのメソッドを呼んだか覚えておく方法で、キャッシュを調べる必要がないので非常に高速である。

ＣＬＯＳの場合には、マルチメソッド、多重継承、メソッド結合を考慮しなければならないのでこれまでより厄介である。しかし、ＣＬＯＳの場合には総称関数毎にキャッシュを持たせられるという有利な点もある（ＰＣＬもそうしている）。これは総称関数毎にキャッシュを持たせると、メソッド名をキーとして使わずにすむからである。

この論文の主要な点は３つある。

（1）ハッシュ表の最適な大きさ
大きさＴのハッシュ表にｎ個の要素が登録されている時、ハッシュ表を何回引けば値が求められるかは、

$$\frac{T+1}{T-n+1}$$

の関係で与えられる。ハッシュ表を大きくすると引くのは高速になる（１回で引ける確率が高くなる）が、メモリをたくさん消費する。このタイムとスペースのコストを１：１の割合いと考えた場合は、

$$T = 2n-1$$

にすると最適である。

（２）クラスの識別コードの提案

ＣＬＯＳの各クラスは、それぞれ特有のクラス優先順位リストを持っている（クラス優先順位リストの先頭はそのクラス自体であるから当然であるが）。クラス優先順位リストはメソッドの継承順位を決定するから、クラス優先順位リストをエンコーディングしたものをクラス識別コードにすればメソッドサーチにおいても使えるだろうというのが考えである。クラス優先順位リストを逆にしたものが

(T STANDARD-OBJECT C0 C1 C2)
(T STANDARD-OBJECT C0 C1 C3)
(T STANDARD-OBJECT C2 C0 C4 C5)
(T STANDARD-OBJECT C2 C6)

であるようなクラスC2、C3、C5、C6があった時、識別コードはそれぞれ（１１１１１）、（１１１１２）、（１１２２３１）、（１１２３）とする。つまり、各リストのｊ番目の要素だけに着目し、その列に出現するクラスを識別する値を与える。実際の表現では、列ではなくメモリブロックを与える。この論文では、列内で異なるクラスの最大を２５５とし、クラス優先順位リストの長さの最大を１６とした、１バイト＊１６（４ワード）で良いとしている。

（３）継承したメソッドのサーチ

(defmethod foo ((x C1) (y C2)) ...) のように定義されたメソッドが呼び出されるのは、引数がC1とC2のインスタンスの場合だけでなく、C1のサブクラスのインスタンスとC2のサブクラスのインスタンスの場合もある。従ってC1とそのサブクラスの個数をｍ、C2とそのサブクラスの個数をｎとすると、このメソッドに限ってもｍ＊ｎ個がキャッシュには登録される可能性がある。継承したメソッドも同じキャッシュに入れることにすると、ハッシュ表を相当大きくしなければならずまたキーの衝突による効率の低下も心配される。

そこで、この論文ではクラス優先順位とラムダリスト上の順番を３分木（ternary trees）を使ってメソッドの継承関係を表現する方法を提案している。たとえば、

(defmethod baa ((x C1) y z) ..)
(defmethod baa ((x C2) (y D1) (z E1)) ..)
(defmethod baa ((x C2) (y D1) (z E2)) ..)
(defmethod baa ((x C2) (y D2) (z E1)) ..)
(defmethod baa ((x C3) (y D1) z) ..)

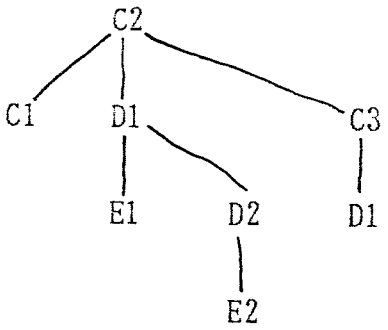
というメソッドがあった時、クラスC、D、Eのインスタンスc、d、eが引数として与えらたとする。またクラスC、D、Eのクラス優先順位が

(C .. C1 .. C2 .. C3 .. t)

(D .. D1 .. D2 .. t)

(E .. E1 .. E2 .. t)

であるとする。この時、総称関数 baa に次のような３分木があればメソッドサーチが容易にできる。



結論： 継承していないメソッドについてはキャッシュを用い、継承したメソッドは３分木を使うとよい。

TICLOS : A high performace implementation of CLOS for the Explorer family

Patrick H. Dussud

September 9, 1988

1 目標

TICLOS を設計するに当たって以下の目標を設定した。

●X3J13 で規定されている CLOS を完全に実現する。

●商用にできるほど高い実行性能を持つようにする。その性能評価は Flavors のコードを CLOS で書き換えて、Flavors で実行するよりも早く実行できるかどうかで行なう。

●低レベルのデータ表現を Flavors と互換性を持たせる。これにより、記憶領域管理のためのシステムソフトウェアやその他の様々なユーティリティを修正する必要がなくなる。

●Flavors との高い互換性を持たせる。少なくとも Flavors は CLOS の一種のメタクラスであるべきで、そうすることにより CLOS のメソッドを Flavors のクラスで特定化することが可能になる。

2 設計哲学

上記の目標を設定し、Flavors システムにおける私達の経験を基にして CLOS は設計された。この設計は Flavors 風のオペレーションに優先順位を与えたものである。

●実行時に必要な情報を予め計算しておくことにより、メソッドやクラスの生成や修正を犠牲にして、実行を速くする。

●一引数のメソッド選択を速くする。多重メソッドやインディビジュアルメソッド選択はこの基本的な機構を複数回適応して行なう。

●データ構造が大きくなることの負荷を犠牲にして、インスタンスの参照を速くする。

もう一つの設計目標はマイクロコードによる特別なサポートを最小限に抑えることであった。処理系はハードウェアから得られる恩恵に預かれない限り、マイクロコードではなく Lisp で実現されるべきである。そしてマイクロコードによる特別なサポートは Flavors の時よりも極めて少なくなった。この設計哲学により私達は格固とした出発点を得た。すなわち私達は Flavors から実行性能の統計を得ることができる。この設計のシミュレーションは実際のプログラム測定から得られるデータでテストすることができる。

3 TICLOS の実現

3.1 インスタンスとクラスの表現

3.1.1

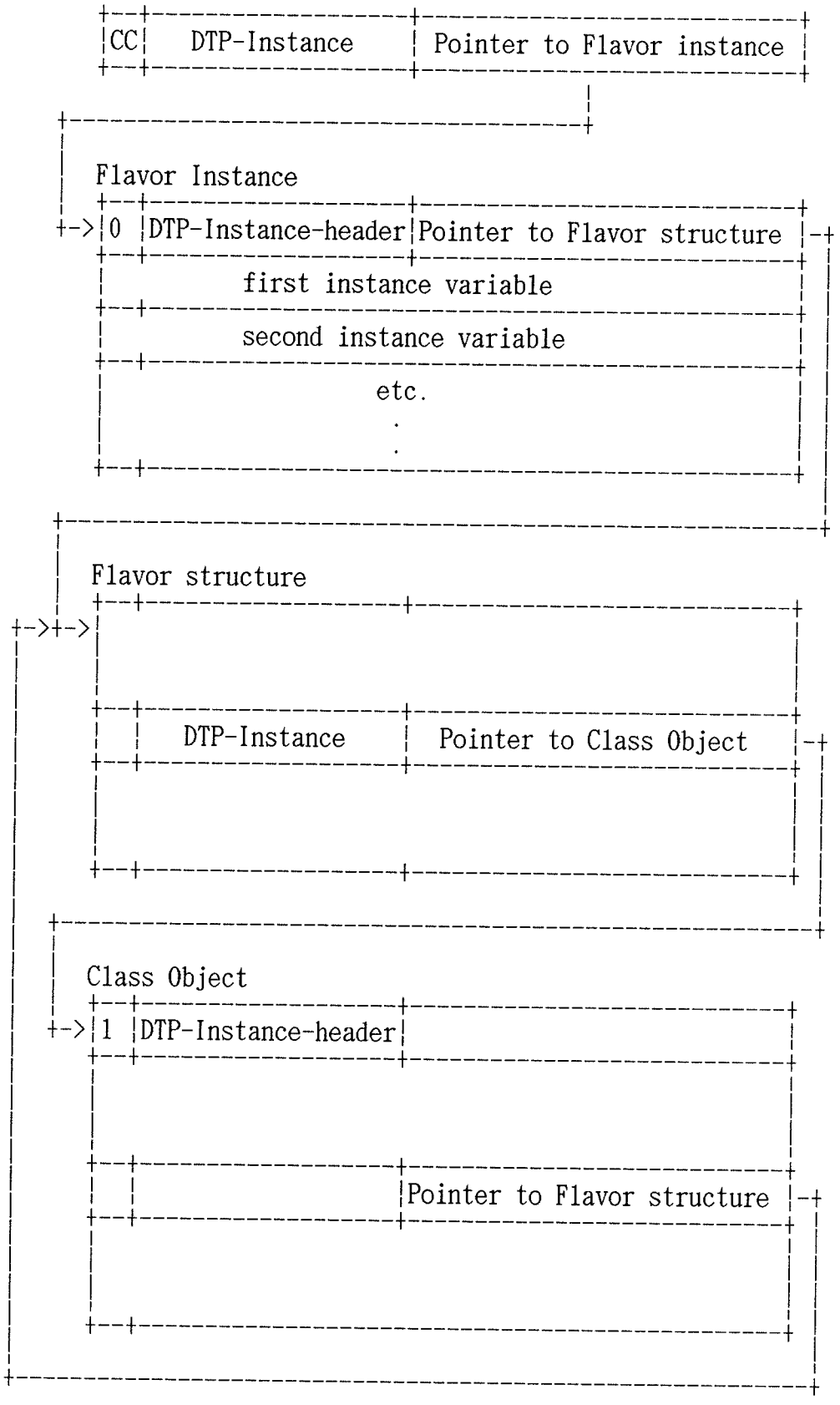
Standard-class オブジェクトは二つの部分に分けられる。クラスオブジェクトは ANSI 標準の第３章で規定されているアクセサを用いて参照できる情報を含むクラスを記述した構造体を指している。これにより、クラスの再定義が可能になる。

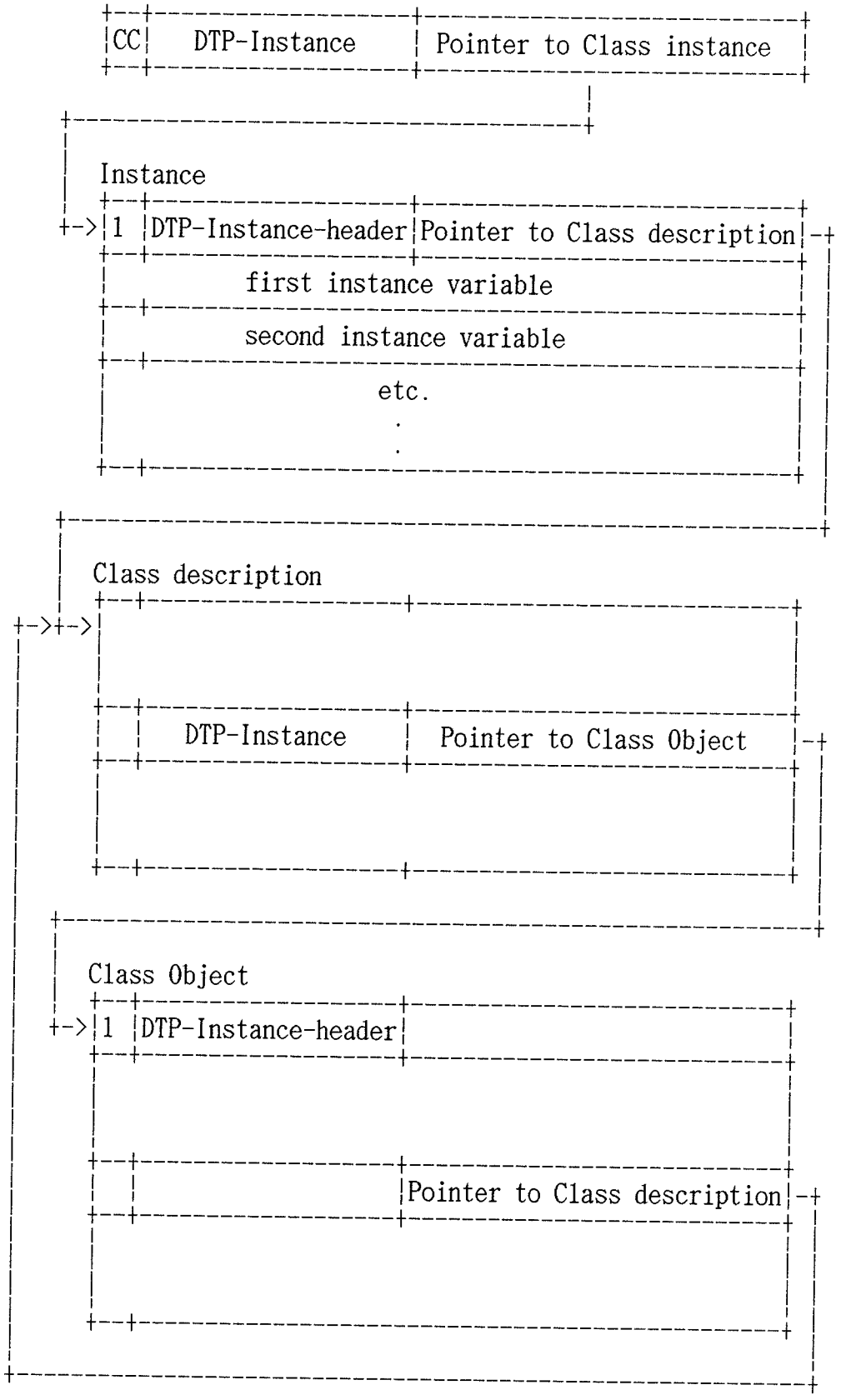
3.1.2

インスタンスの最初のワードはタグとそのクラス記述を指すポインタである。そしてその後にすべてのインスタンススロットが続く。これは Flavor 構造体と呼ばれるクラスを記述する構造体を持つ Flavors に非常によく似ている。これによりゴミ集めを修正することなしに CLOS を実行することができる。TICLOS のインスタンスはヘッダワードのビット（2ビットある cdr コードの上位ビット）によって Flavors のインスタンスと区別される。

3.2 総称関数の表現

総称関数は通常のコンパイルされた関数（FEF）として表現され、そのデバッグ情報スロットが総称関数のスロットを含む構造体を指している。これにより、


CC
DTP-Instance
Pointer to Flavor instance
Flavor Instance
0
DTP-Instance-header
Pointer to Flavor structure
first instance variable
second instance variable
etc.
Flavor structure
DTP-Instance
Pointer to Class Object
Class Object
1
DTP-Instance-header
Pointer to Flavor structure


CC
DTP-Instance
Pointer to Class instance
Instance
1
DTP-Instance-header
Pointer to Class description
first instance variable
second instance variable
etc.
Class description
DTP-Instance
Pointer to Class Object
Class Object
1
DTP-Instance-header
Pointer to Class description

Translation of KEE Object Oriented Programs into CLOS-XT

A.Dappert-Farquhar
J.Estenfeld
Siemens AG, FRG

1. 概要

Siemens で開発された KEE のオブジェクト指向プログラミング機構を用いたアプリケーションを CLOS の拡張である CLOS-XT に変換する話.

2. CLOS の拡張

KEE UNITS のユーザのサポートのために有益な機能を CLOS に取り入れ, CLOS-XT を開発した. これらの機能は KEE の知識ベースを変換するためにも有効である.

- CLOS のスロットオプションにインヘリタンスオプションを加えた. これは, 継承を終了するために, または, ユーザ定義の継承を利用できるようにするものである. また, 各スロットにドキュメンテーションオプションを加えた.
- defclass のスロットオプションを拡張し, 付加的な記述やユーザ定義の注釈などを追加できるようにした. 例: name, initform, documentation.
- インスタンスをオブジェクトと呼べるようにした. それぞれのクラスはそのインスタンスを知っている.
- クラスとインスタンスは知識ベースに適用され処理を受ける.
- クラスとインスタンスの操作(クラスの削除, あるクラスの知識ベースの変更, 新しいスーパークラスの追加など)と情報を得るための様々な機能を与えた.

3. KEE の知識ベースから CLOS-XT への変換

KEE から CLOS-XT への変換には, 次の 2 つの仕事がある.

(1) KEE の知識ベースファイルから CLOS-XT のファイルへの変換

(2) KEE の関数の変換

3.1 KEE の知識ベースファイルから CLOS-XT のファイルへの変換

ここでの主な問題は次の 2 点に置ける概念の違いである.

- KEE のユニットと CLOS のクラスとインスタンス
- KEE の own, member スロットと CLOS のインスタンススロット

現在は, ユニットを表すリスト構造を知識ベースの定義, クラスとインスタンスに変換するアルゴリズムを実現している. superclass.parent.list, member.of.parent.list, member.slot.list をパージングすることにより, 知識ベース, 実測値, インスタンスとして扱うことのできるユニット, クラスとして扱うことのできるユニットに分類する. own スロットとその継承があるかどうかによって, 任意のメタクラスが生成されなければならない. KEE のメソッドはクラスのスロットの中にはないが, CLOS のメソッドに変換される. 実測値もメソッドになる.

3.2 KEE の関数の変換

ここでもユニットを参照する KEE の関数の変換が問題となる．これは関数の引数のバージングによって行なう．このモジュールの仕事はまだ始まったばかりで，動的な振舞いなど問題が残っており，先行きは分からない．

4. 結論

CLOS-XT はメタオブジェクトプロトコルレベルのものであり，変換は CLOS-XT を使って実現されている．完全な自動変換は不可能で，KEE の全ての機能を CLOS のほとんどの特徴に手を加えないまま綺麗に変換するのは無理があるようだ．これは，もっともなことでもなければ，必ずそうであるということでもない．したがって， KEE UNITS を使った開発に制限を加えて，手作業の変換が必要な場合はファイルに記述する必要がある．この制限は，例えば，１つのインスタンスが複数のクラスに属すること(多重の member-parent-link)を禁止する，スロットの値を１個だけに制限することである．しかし，自動変換が有効なサポートになり CLOS-XT を利用するユーザが増えることを願っている．CLOS-XT と変換プログラムは，会社のプロダクトにする計画はなく，内部の目的のための利用を意図している．

Common Lisp Object System に対する注意

リチャード ガブリエル
ルシッドインコンポレーションとスタンフォード大学

ガブリエルの論文の要約を以下に述べる。

プログラム言語の標準化はその言語の進歩や研究を凍結してしまい、さらにその言語が最新であればその言語の経験を組み込むことができないから危険である。

CLOS は他の言語と比較してメソッド継承のかわりにメソッド適用(applicability)を使うところが新機軸である。CLOS の設計方針は主に NewFlavors や CommonLoops から得ているが、それらの言語はメタオブジェクトのシステムを持っていない。

CLOS は数少ない先進的なオブジェクト指向言語の一つであり、CLOS ライクな言語がオブジェクト指向の研究の最前線になろう。

1．メソッド継承からメソッド適用への置き換え

メソッド継承は単一操作に対してうまく働くが、複数操作に対してはうまく働かない。メソッド適用はメソッド継承の一般化であるというが、この一般化はメソッドが第一引数のみに適用されるのではないということだけである。

総称関数の良い実現はイベントやアクションが起こればいつでも複数の引数の総称関数を呼び出すことである。サブクラスやそこに送るメソッドに対するクラス階層を拡張することにより、エレガントな方法でシステムを実現できる。デリゲーション(delegation)は別のオブジェクト指向のモデルである。その利点はこれらのカテゴリのモデル化が行なえ、既定値を自然に扱うことができ、あの SmallTalk のマジック「self」の問題がないなどがある。しかしデリゲーションは CLOS に比べて弱い。その理由はデリゲーションがいろいろなレベルを混合させているからである。インヘリタンスシステムにおいて、メソッド作成者は他のメソッドなどを気にせずに書けるがデリゲーションにおいてはオブジェクトやメソッドの特性などに気を付けなければならない。

2．複数の領域からの継承

「多重継承」と言う言葉は大袈裟すぎる。多重継承において衝突が起こったときはシャドウ、エラー、結合の方法とどのメソッドやスロットを選択するかのメカニズムをプログラムに与える方法によって解決する。CLOS ではすべての衝突の扱いを一様には扱えず、メソッドでは結合が使われ、スロットでは継承、シャドウ、結合が使われる。シャドウの実現のために、クラス優先リストが計算され、単一継承のシャドウがこのリストを使う。これはうまく働くが、エレガントな実現ではない。

3．メソッド衝突に対するメソッド結合

メソッド結合は CLOS における興味のある概念の一つである。私が CLOS を設計するなら、同時に２個以上のクラスのインスタンスであるようなインスタンスを与える。それを「多重一単一継承」と呼ぶ。このアプローチでは不自由さをなくすためにメソッド結合のもっと手の込んだ使用をする。

CLOS の中で興味を持つアイディアは実効メソッドがプログラマが直接プログラムしたものではなく、実行時にそのコードが生成されることである。CLOS のメソッド結合は全く見苦しく、閉包を使っての実現はつまらないと思う。そのようなものは他にどうしようもなく作成した計算モデルの一つである。

４．言語のすべての見方のファーストクラス扱い

CLOS は他のオブジェクト指向言語よりは多くのタイプをファーストクラスとするがすべてはないであろう。メタオブジェクトプロトコルはファーストクラスにすることが可能である。CLOSはそれを十分にはサポートしていないが使用できる。

５．初期化

CLOS は他の言語よりクラスの初期化、再定義に関して進んでいるところがある。総称関数もメソッドも CommonLisp のラムダリストを使う。設計者がラムダリストの選択をしたのは関数定義のようにメソッド定義ができると思ったからである。総称関数とメソッドのインタフェースがすっきりすれば文法の混乱はなくならないが初期化プロトコルは簡単になる。総称関数は結合メソッドに対するインタフェースであるということをさらに押し進めることが重要である。

６．まとめ

CLOS は通常のオブジェクト指向プログラム言語上にうまくのっている。CLOS ライクな言語が続くならば、オブジェクト指向システムの研究に尽きぬ源泉となろう。

以上がガブリエルの論文の要約であるが、全体的に CLOS をもろ手を挙げて歓迎しているのではなく、一つの新しいオブジェクト指向言語として他のシステムと比較して長所と欠点（特に欠点）を述べている。長所は具体的に書かれているが、欠点はどちらかといえば抽象的である。これは最初にも書かれていたように CLOS の経験（特にメタオブジェクトシステム）の経験が少ないからではないだろうか。我々はメタオブジェクトシステムについて十分な経験を積んでから、その評価（CLOS 全体の評価も含む）をしなければならないと思う。メソッド適用はガブリエルも言っているように CLOS の最も大きい新機軸であると思う。複数引数の操作を使いこなせば新しいプログラムになる。またガブリエルも述べているように CLOS ライクな言語がこれからのオブジェクト指向言語の研究の最前線になるであろう。