Abstract

This paper introduces a new formalism, Constraint-Based Invocation, that combines the Object-Oriented and Rule-Based paradigms in an elegant and orthogonal way.

The Constraint-Based model is a generalization of traditional Object-Oriented paradigms based on three orthogonal sub-paradigms:

1. **Constraint-Based Invocation**: A generalization of the traditional method invocation where dispatch is done based on the types of the arguments. In Constraint-Based Invocation dispatch is done based on constraints that are arbitrary user defined predicates.

2. **Instance Inheritance**: A dual to the concept of Class Inheritance in the sense that class inheritance structures classes and Instance Inheritance structures instances.

3. **Procedural Attachment**: Also known as active values or access-oriented programming, wherein a function is called in a data-driven manner. The semantics of this concept are generalized to all objects in the Constraint-Based model.

An additional central philosophical argument of the paper is that so-called 'multi-paradigm' languages should be developed not by combining paradigms in a partially integrated system, but rather by their synergistic unification under a new, subsuming paradigm.

Therefore, the top level design goals are to show:

1. that Object-Oriented and Rule-Based programming paradigms are merged in a synergistic way under the Constraint-Based paradigm,

2. that the combination results in a system for which performance improvements over existing systems may be demonstrated, and

3. the semantics of the Constraint-Based paradigm may be defined in a formal way.

The Object-Oriented and Rule-Based paradigms appear to have complementary strengths and weaknesses (See Figure 1-1), but this, of course, does not guarantee that their combination will result in a system demonstrating only the strengths of each. The idea of combining Rule-Based and Object-Oriented paradigms is certainly not new. There have been a number of rather successful attempts. The resulting systems are extremely powerful, and we offer them as an existence proof that the combination is, in fact, promising. These attempts suffer, however, from their lack of a single underlying model of computation, as can be seen from their size, complexity and performance [13].

Examples of such systems are: Loops [15], KEE [7, 11], ART [4] and many others.
Recalling the idealism of the late 60's this attempt tries to design a paradigm:

1. out of a few orthogonal constructs and
2. whose semantics may be described in a formal way.

1.1. Terminology and Outline

In the sections that follow, the Constraint-Based paradigm is compared to a number of other paradigms. Numerous examples are given in the language CBL, which is the first instance of this new paradigm. The semantics of the paradigm are presented informally. A formal semantics of the paradigm appears in (18). CBL is built upon the Common Lisp Object System (CLOS) [1], which turned out to be a particularly convenient platform. The CLOS system is in turn built upon Common Lisp [14]. Another possible platform would be that of C++ [17] or a language could be built from the ground up. In general, the notation used is borrowed from "Common Lisp: The Language" [14] and "The Common Lisp Object System Specification" [1].

The following definitions are central to an understanding of the Constraint-Based paradigm. They appear in full detail in [1], but are abstracted here:

- A class is an object that determines the structure and behavior of its instances.
- A class is an instance of the metaclass standard-class.
- A class that inherits structure and behavior from another class is a subclass of the class from which it inherits.
- A generic function is an object whose behavior is defined by the methods from which it is composed.
- A method is an object that contains a method function that implements its behavior and a sequence of parameter specializers that specify when the method is applicable.
- A method is an instance of the class standard-method.
- A method function is the function that implements the behavior of its method.

The section that follows is a digression on what the term performance means in the context of paradigms design and evaluation. Section 3 describes the three orthogonal concepts from which the Constraint-Based paradigm is composed. Section 4 provides a proof of the orthogonality of these concepts. A small example of the encoding of Conway's game of life is then given in section 6 and the performance improvements that result are described. Finally, there is a brief discussion of some related works and concluding remarks.

2. Digression on Performance

The designer of a new programming paradigm is faced with a number of difficult tasks, but one stands out as being the most difficult. This is the task of making a convincing argument that the newly designed paradigm is in some way better than that which already exists. The difficulty of task stems directly from an imprecision of definition "better". The software engineer, unlike the hardware engineer or the algorithm designer, is at a great disadvantage because of the lack of clarity of the term "better" in his vocabulary. The hardware engineer has quantitative metrics such as: the number of gates, the speed of the circuit, or the area of silicon. The theorist has metrics such as: the worst case complexity, the best case complexity, or the average complexity. Various metrics have been proposed for software, but they are highly qualitative, and the software engineer is often found using such ethereal measures as: more expressive, easier to program in, or even more natural.

Thus to prove the assertion that a given language is "better" than another, the language designer must study the diversity of men along with the diversity of paradigms (carefully factoring out the human diversity). This necessitates a large random sample and the use of the statistical methods of the social sciences. Such studies are lengthy, very expensive and usually beyond the abilities of the most academic and even the industrial research organizations. Even if a study is carried out, ultimately a paradigm is only deemed successful if it has survived sufficient time and use. Judging from such languages as Pascal [21], C [12] and Smalltalk [9] "sufficient time" appears to be in the range of 10 to 15 years.

How then can the language designer hope to make convincing claims about new designs? A paradigm may be evaluated in terms of its performance. There are two aspects to performance in language design. The first is that of expressiveness, ease of use, modifiability and other fuzzy terms. Let us call this aesthetic performance. The other aspect of performance is that of compilability, and run-time efficiency. Let us call this pragmatic performance. Making an argument about aesthetic performance is the much harder of the two. It is difficult if not impossible to quantify without the kind of factoring described above. The designer's only hope of

---

2The intent of this figure is not to indicate that one paradigm has or does not have a particular characteristic, but rather to indicate that that characteristic is more strongly supported by that particular paradigm.

3It is important to note that a method is an object.
making a convincing argument is to demonstrate improvements in the areas of universally accepted truisms of engineering design such as: orthogonality, modularity, and abstraction. To follow this path it is necessary to present several well known algorithms encoded in the new formalism, and demonstrate convincingly that at least one of the above truisms is enforced without sacrifice in the others.

The other aspect of performance, pragmatic performance, is much easier to deal with since it is possible to give quantitative measures of both compilability and run-time efficiency of a new language. Again, this is best done through the presentation of examples and the measurements generated from the examples. Care must be taken to insure that differences in implementation are either factored out or taken into account.

The Constraint-Based paradigm will be evaluated in terms of the definition of performance given above (See Figure 2-1).

3. The Paradigm

The Constraint-Based paradigm may be considered the result of the combination of three sub-paradigms. The first is the extension of Object-Oriented method invocation to include programmer defined predicates. The second is a new concept of instance inheritance and the third is that of procedural attachment [16].

3.1. Constraint-Based Invocation

Constraint-Based invocation differs from the traditional notion of method invocation [9]. In a traditional Object-Oriented language, when a generic function is invoked the actual method dispatched to is determined solely by the types of the actual arguments. In the Constraint-Based paradigm the method dispatched to is determined instead by constraints that have been placed on the arguments. A constraint is an object that represents the boolean combination of programmer-defined predicates. The constraints of a method, therefore, constitute the necessary preconditions for the execution of the method. This extends the semantics of method invocation such that it includes the semantics of traditional Object-Oriented languages and allows constrained-methods to be used in a manner similar to that of rules in Rule-Based languages. In order to make this clear we examine the essence of method invocation in relation to the semantics of rules.
The correction of the GCD function above illustrates a very important quality of methods as opposed to procedural or functional programming. This quality may be characterized as that of stepwise refinement or evolution. A system where a function is defined by the composite behavior of a group of independent methods allows the semantics of a function to evolve incrementally over time through the addition of methods. This has been recognized as an important quality in existing Object-Oriented languages [19], and the Constraint-Based paradigm extends its applicability from types alone to methods. In addition, the constraints of a method give the programmer and the system an additional semantic handle by which to access and manipulate methods and the generic functions they compose.

By allowing the boolean composition of n-ary predicates as its dispatch mechanism, Constraint-Based invocation represents the generalization of traditional method invocation in such a way as to subsume the semantics of both rules and methods. We have in this examination (as some readers may have noticed) overlooked a very important part of Rule-Based semantics. Rules, unlike methods are not explicitly invoked, but rather are invoked in a data-driven fashion by changes to a global database. Thus, we have only captured part of the Rule-Based semantics by our generalization of method invocation. The remainder of the semantics will be captured by the two other concepts of the Constraint-Based paradigm described in the following two subsections (3.2 and 3.3).

3.2. Instance Inheritance

The Object-Oriented paradigm is further extended with a new concept, that of Instance Inheritance. Instance Inheritance is the concept of inheritance between collections of instances in contrast to program external. Object-Oriented inheritance which inheritance of structure and behavior occurs between classes. Instance Inheritance is implemented in the form of collections. A collection is an unordered grouping of objects (instances) which may be restricted to contain objects that satisfy certain constraints (a constrained collection). Collections may serve to structure a single type, or types of objects. For example, in the case of multi-methods which, unlike traditional methods, do not "belong" to a single class, collections may be used to group them together in a semantically meaningful way. Used in this manner a collection may be viewed as taking the place of a module facility.

The utility of collections is not limited to multi-methods. Collections allow the expression of abstractions that are very difficult without a notion of instance inheritance. The claim has been made that one of the great advantages of the Object-Oriented paradigm is that it allows the definition of a hierarchically structured abstraction space [Wegner87].

1. Inheritance structures Classes.
2. Classes structure Objects.
3. Objects structure operations and data.

This is in fact true, but it does not mean that all hierarchically structured abstractions are easily expressed. For example, suppose a programmer wishes to express the concept "Made in Japan". If she is only dealing with a single type of object -- for example cars -- she may express it nicely by forming the hierarchy given in Figure 3-1.

If, on the other hand, she has more than one type of thing that she is working with she has two choices. These are shown in Figure 3-2. If the programmer chooses the first option she must define individual notions of 'made in Japan' for each of the cars, TV, and chip, but this fails to capture the composite notion. If our programmer chooses the second option, she captures the composite notion of 'made in Japan', but she is forced to duplicate the definitions of the individual items. Neither of these solutions is terribly pleasing.

If the concept of Instance Inheritance is introduced into the paradigm, the abstraction "Things Made In Japan" can be simply expressed as a constrained collection:

\begin{verbatim}
(defcollect TMJ (thing)
  (:constraint thing (made eq Japan)))
\end{verbatim}

An argument can be made that this concept could be expressed in terms of either a multiple inheritance hierarchy or mixins. This is true, but a collection embodies the additional semantics of a declarative, but dynamically, system-maintained, data-structure.

Instance Inheritance improves modularity and abstraction and allows additional semantics to be captured within the Object-Oriented paradigm. Collections have a great number of possible uses. Many of these uses are found in other languages, but they are frequently provided by separate extrasyntactic/semantic additions to the language rather than being a integral part of it. Examples include the package system of Common Lisp and the module facilities of a number of other languages [17, 22] as well as the concept of views in database systems [2] and worlds in inferencing systems [20]. Unfortunately, space does not permit an examination of these applications of Instance Inheritance.

4There remains another bug in our gcd function. Its identification and repair is left as an exercise for the reader.

5The fact that collections are unordered is important in the definition of parallel execution semantics, but will not be discussed in this paper.

6For example, consider a system that assumes all Toyotas are made in Japan, but later discovers the existence of an American plant and updates the instances appropriately. The desired operation is that the collection TMJ should maintain its semantics by adjusting its membership through such a change.
3.3. Procedural Attachment

The last concept that is missing to complete the subsumption of Rule-Based semantics is that of procedural attachment. This is needed because, although collections provide the databases, they do not provide the data-driven semantics of the Rule-Based paradigm.

Procedural attachment is not a new concept. It has appeared both in Frame-Based and Object-Oriented systems. It is also known as access oriented programming and may be defined as the concept whereby a generic function may be attached to another object in such a way that it is invoked when that object is accessed. It is a generalization of the earlier concept of daemons. In most systems what is actually provided is the ability to attach a generic function to a particular slot of a particular object. In the Constraint-Based model this is generalized so that a generic function may be attached to:

1. a Slot
2. an Object
3. a Class
4. a Collection

Since all of the above are objects, the generalization claimed above is not a generalization of the concept of attachment, but a generalization of the way that concept has been implemented in existing systems.

This concludes the presentation of the three orthogonal features needed to subsume the Object-Oriented and Rule-Based paradigms under the Constraint-Based model. The next section presents a proof of the orthogonality of these concepts and explains their semantics when taken in combination.

4. Proof of Orthogonality or (3 Choose 2)

4.1. Constrained Methods and Collections

The semantics of a constrained method applied to a collection of objects are that, unless the argument of the method is constrained to be a collection, the method is applied to every object in the collection. The result returned is the collection of values returned from the executed methods. For example, the method fourwheel-toyota-truck given below when applied to the collection Things Made In Japan (TMII) returns a collection of four wheeled, Toyota made, trucks:

```
(defcollect TMII (thing) ...)
(defmethod fourwheel-toyota-truck (t)
  (:constraint t (thing)
    (wheels (= 4))
    (make (eq Toyota))
    (type (eq 'truck)))
  t)
```

These semantics provide the associative retrieval found in Rule-Based systems, but in a different form. This form is actually closer to the semantics provided by database query languages.
4.2. Constrained Methods and Procedural Attachment

Rule-Based semantics are provided by attaching constrained methods to objects. For example, the collection Import-Rules below is attached to the collection of Things Made In Japan. Every time an object is added to the TMIJ collection Import-Rules will be invoked. Efficiency may be gained by attaching the rules to only specific slots, or collections of slots, of an object.

(defcollect Import-Rules ...)
(attach-method-collection
 TMIJ
 Import-Rules)

4.3. Collections and Procedural Attachment

The use of this combination has already been seen in the definition of the constrained collection TMIJ. It can be claimed that we are cheating here in that we are assuming the existence of constrained methods to show the orthogonality of collections and procedural attachment. Although this is true, it hardly seems reasonable to speak of procedural attachment without something to attach. Collections need not be constrained. An unconstrained collection has the semantics that it contains whatever is explicitly put in it. Things may also be explicitly placed in a constrained collection, but this will generate an error if the constraints of the collection are violated. The constraints of a collection, therefore, provide a semantic description of its contents and may be used by the system to optimize queries such as those described in 4.1.

5. Resolution

Method resolution is one area where the difficulty of an existing problem is apparently worsened by the increased generality of the Constraint-Based paradigm. In most traditional Object-Oriented systems when problems of method resolution occur due to multiple inheritance, a fixed, global resolution strategy is used. This is possible because it is relatively easy to define a total ordering on the types of the methods of a generic function. This is clearly not possible for constraints. The strategy adopted in the Constraint-Based paradigm is to allow the definition of a constraint hierarchy and to allow the use of meta-methods where the hierarchy is inadequate for complete resolution. The basic idea behind the design of the constraint hierarchy is to try to localize the decision process as much as possible. In terms of the implementation, this means that when two constraints that have the same type are being compared, the type of the constraints is charged with the determination of which is more specific. Similarly, if the two constraints being compared are each composed of the same simpler constraint, then this constraint is charged with the determination. Within each constraint type a constraint precedence list may be given that defines the precedence of each constraint in relation to the others of that type. The definition for the numeric type constraint and an example of the numeric constraint = are given below:

(defconstraint =
 :type 'numeric-constraint
 :drm #'most-specific=))

In the above example :drm stands for the default resolution method. Each generic function may specify its own top level resolution function so that the default strategies may be altered on a function by function basis. Finally, in the cases were the hierarchy is insufficient or a special specific resolution procedure is necessary a meta-method may be specified. An example of where a meta-method is useful is the following:

The constraints equal and member are both top level constraints and have no precedence relation. There should be a rule in the system that states that, if the argument to a equal constraint is disjoint from the list of arguments to a member constraint, the two constraints are exclusive of one another.

This rule is given in CBL as:

(defmethod resolve-same-precedence (ml m2)
 (:constraint ml (method)
  (constraint c1
   (member
    equal-constraint
    member-constraint)))
 (:constraint m2 (method)
  (constraint
   member
   equal-constraint
   member-constraint)
  (neq c1)))
 ... :extract actual or compile-time args
    (if (not (member arg1 args2))
     'exclusive))

A pictorial representation of the entire resolution algorithm is given in Figure 3-1.

6. An Example

In order to demonstrate pragmatic performance improvements of a system, it is important to have a library of independently developed programs that can be recoded into the new paradigm and benchmarked in relationship to their original version. If the conversion can be done automatically and performance improvements can be demonstrated, the argument is even more convincing. If human intervention is involved, the argument is significantly less convincing.

One of the libraries of code that was chosen for benchmarking CBL was a number of OPS5 programs available in the public domain. An automatic converter was written in CBL that converted the OPS5 programs into a form that would execute, but with no performance gain. The converted source was then modified to take advantage of each combination of features described in Section 4. Due to space limitations, only one of the systems will presented here.

---

7CBL has an insert-object method constrained to collections.

8The rules were converted into constrained methods, the working memory into a collection of objects and the constrained methods were attached to this collection. A resolution meta-method was written mimicking the conflict resolution semantics of OPS5.
The system that will be presented here is a simulation of Conway's Game of Life. In the Game of Life one has a grid of cells and the vitality of each cell is determined by a simple set of rules. The algorithm may be simply stated:

For each generation < n

For each cell:
1. Count the number of living neighbors of each cell.
2. Update the cell's state based on the number of living neighbors and the rules of the Game.
3. Print that cell.

Two of the rules of the game are presented in Figure 6-1 and the three collections used are given in 6-2. The code that was machine translated appears in uppercase; modified code is in lowercase.

6.1. Some Results

Two sets of results are presented here. In the first the difference in the number of comparisons between a CBL system without sufficient semantic knowledge of its constraints to order the methods of the system is compared to a system to which sufficient semantic knowledge in the form of added meta-methods (Table 6-1). In the second set of results the overall improvement by recoding the system is presented (Table 6-2).
(defcollect cells ()
  ((make-cell :n-made 'no :generation '0
             :neighbors '0 :status 'dead
             :x '1 :y '1)
   (make-cell :n-made 'no :generation '0
             :neighbors '0 :status 'alive
             :x '2 :y '1) ...)
)

(defcollect acells (cells)
  (:constraint cells (cell) (status (eq 'alive))))
(defcollect dcells (cells)
  (:constraint cells (cell) (status (eq 'dead))))

7. Conclusion

The Constraint-Based paradigm has been presented as the composition of three orthogonal constructs: Constraint-Based invocation, Instance Inheritance, and Procedural Attachment. Improvements, both aesthetic and pragmatic, that result from applying this model of computation to the merging of Object-Oriented and Rule-Based paradigms have been presented. Although not described in this paper, the semantics of CBL may be described in CBL. This means of formal specification was found superior to others because a self specification may be executed, tested, and debugged.

The similarity of constrained methods to Dijkstra's guarded commands [6] has most probably already been noted by many readers. This similarity is not accidental in the sense that the Constraint-Based paradigm serves to provide the programmer with languages in which he may more easily express his algorithms and with a language that provides the axiomatic semantics of its own definition.

An effort was made to make as few changes as possible to the existing code in modifying it to run in CBL. A version of Life written without this effort ran over 200 times as fast as the original version which at least demonstrates that some restraint was used in the recoding.

<table>
<thead>
<tr>
<th>Number of Comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Version</strong></td>
</tr>
<tr>
<td>Life1</td>
</tr>
<tr>
<td>Life2</td>
</tr>
<tr>
<td>Life3</td>
</tr>
</tbody>
</table>

Table 6-1: Ordered vs Unordered Methods

The second set of benchmarks of overall performance improvements is less interesting than the first. The second set does indicate, however, that the Constraint-Based paradigm offers a path of improvement from Rule-Based to Object-Oriented systems if an algorithm or parts of it are coded in the wrong paradigm to begin with. This path is very important, because work in the expert-systems area has shown that it is, in fact, a path frequently traversed over the course of the development of a real system. This is not surprising considering that the acquisition of expertise may be viewed as the movement from search toward heuristic and then often to algorithmic solutions. This refinement ability is missing in other systems.

As mentioned above, three other small systems were benchmarked, all with similar results. Larger systems are currently being converted and benchmarked and these results will be reported in future work [18].

---

9As is probably the case with the current example.
# Number of Comparisons

<table>
<thead>
<tr>
<th>Version</th>
<th>Number of Compare's</th>
<th>CPU Time</th>
<th>Improvement Compare's</th>
<th>Improvement CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPS5</td>
<td>141042</td>
<td>10:16.94</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Life1</td>
<td>81748</td>
<td>2:16.48</td>
<td>42%</td>
<td>4.5X</td>
</tr>
<tr>
<td>Life2</td>
<td>55877</td>
<td>2:45.76</td>
<td>60%</td>
<td>3.7X</td>
</tr>
<tr>
<td>Life3</td>
<td>11643</td>
<td>:13.32</td>
<td>91%</td>
<td>46.3X</td>
</tr>
</tbody>
</table>

### Table 6-2: Overall Improvement

## References


