

## 4. Constraint Programming Systems

### 4.1 Kaleidoscope: A Constraint Imperative Programming Language

Gus Lopez, Bjorn Freeman-Benson, and Alan Borning

Gus Lopez and Alan Borning  
Dept. of Computer Science & Engineering, FR-35  
University of Washington  
Seattle, Washington 98195  
USA  
{lopez,borning}@cs.washington.edu

Bjorn Freeman-Benson  
School of Computer Science  
Carleton University  
Herzberg Building  
Ottawa, Ontario K1S 5B6  
CANADA  
bnfb@scs.carleton.ca

#### Abstract

The Constraint Imperative Programming (CIP) family of languages integrates constraints and imperative, object-oriented programming. In addition to combining the useful features of both paradigms, there are synergistic effects of this integration, such as the ability to define constraints over user-defined domains. We discuss characteristics of the CIP family and provide a rationale for its creation. The synergy of constraints and objects imposes additional challenges for the provision of constructs, such as object identity and class membership, that are well-understood in conventional language paradigms. We discuss the benefits and challenges of combining the constraint and imperative paradigms, and present our current ideas in the context of the design and implementation of the Kaleidoscope'93 language.

#### 4.1.1 Introduction

Imperative programming languages are relatively well understood, used by a large number of programmers, and supported by numerous software tools. However, these languages are often more low-level than one would like. Focusing on

**Table 4.1.** Imperative versus CIP Code

<i>Imperative</i>	<i>Constraint Imperative</i>
<pre> while mouse.button = down do   old ← mercury.top;   mercury.top ← mouse.location.y;   degrees ← mercury.height/scale;   if old &lt; mercury.top then     delta_grey(old,mercury.top);     display_number(degrees);   elseif old &gt; mercury.top then     delta_white(mercury.top,old);     display_number(degrees);   end if; end while; </pre>	<pre> always: degrees = mercury.height/scale; always: white_rectangle(thermometer); always: grey_rectangle(mercury); always: display_number(degrees); while mouse.button = down do   mercury.top = mouse.location.y; end while; </pre>

interactive graphical applications, we examined a number of user interfaces and observed that some portions are most clearly and conveniently described using constraints—relations that should be maintained—while other portions are most clearly described using standard imperative constructs. However, none of the imperative languages used to program these interfaces directly supported constraints. Thus, the constraints were encoded implicitly, by hand, and each constraint was enforced by a widely distributed set of small code fragments—a recipe for maintenance headaches.

To address this problem, we proposed constraint imperative programming (CIP), an integration of two disparate paradigms: a standard object-oriented imperative one, and a declarative constraint one (Freeman-Benson 1991; Freeman-Benson and Borning 1992a; Freeman-Benson and Borning 1992b). Compare the two code fragments in Figure 4.1, which allow the user to drag the mercury of a thermometer up and down. The version on the left uses only standard imperative constructs. It requires the programmer to check whether values have changed and if so, to fill or erase the appropriate rectangle increment and then redisplay the temperature value. The constraint version on the right uses a combination of imperative constructs and constraints. Some of the constraints specify relations that must always hold (e.g. `temperature = mercury.height / scale`), while others specify relations that should hold only while a given condition is true (e.g. the constraint `mercury.top = mouse.location.y`, which holds only when the mouse button is down). Imperative constructs, such as the `while` statement, are used to control program execution, in particular, when certain constraints should hold.

Consider an object, `VerticalLine`, represented as two points. If this object were implemented in an imperative language, all operations on `VerticalLines` would have to ensure that the `x` fields of both points were equal. Since there

is an implicit integrity constraint that these  $x$  fields remain equal, it is the programmer's responsibility to maintain this constraint. A CIP implementation of `VerticalLine` would simply assert this constraint when the object was created (e.g. `always: p1.x = p2.x`) and maintain it automatically for the programmer.

There are advantages to CIP over approaches based in logic programming: in particular, CIP languages support mutable objects and object identity in a way that is familiar to current users of object-oriented programming. Due to the popularity of imperative programming (and object-oriented programming in particular), CIP offers an evolutionary mechanism (or maybe a Trojan Horse?) for introducing constraint programming.

We have found it useful to extend the constraint paradigm to allow both required and nonrequired (or preferential) constraints. The required constraints must hold for all solutions, while the preferential constraints should be satisfied if possible, but no error condition arises if they are not. A constraint hierarchy can contain an arbitrary number of levels of preference (strengths). The original motivation for this extension was to give a declarative semantics for what to change when perturbing the values in a constraint system. (For example, suppose we have a constraint  $A+B=C$ , and edit the value of  $B$ . Should we change just  $A$ , change just  $C$ , change both  $A$  and  $C$ , undo the change to  $B$ , or what?) However, constraint hierarchies have additional uses for expressing preferences in planning, layout and other domains. (See (Borning, Freeman-Benson, and Wilson 1992) for more information.)

In addition to constraints of varying strength, Kaleidoscope has constraints of varying duration. Constraint durations specify the period of validity for constraints. The most flexible model would allow constraints to be asserted and retracted at arbitrary points in time. However this would lead to difficulties in predicting behavior, since any piece of code could side-effect which constraints are active. Instead, in our design, the default constraint duration is `always`, which causes a constraint to remain active forever. A `once` duration instructs the system to assert the constraint, causing it to be enforced at that moment (and thus potentially affecting values), and then immediately retract it. Finally, the `assert/during` construct specifies that a constraint should remain in force during the execution of a block or loop. We might make an analogy with the GOTO statement/structured programming controversy of the 60's: GOTO statements are analogous to constructs that allow constraints to be asserted and retracted at arbitrary times, while structured control statements are analogous to the control structures in Kaleidoscope.

Kaleidoscope'93 is a class-based object-oriented language with multi-methods. In a conventional object-oriented language, the method that is executed in response to a message is determined purely by the receiver of the message. For example, if we send the message `display` to a circle object with a bitmap as an argument, the method to be executed is determined solely by the circle. In contrast, in a language with multi-methods, all arguments potentially may participate in selecting the method; thus in the preceding example, both the circle and the bitmap might participate in the choice of method, allow-

ing a different method to be invoked when displaying on a vector display. The best-known language that supports multi-methods is the Common Lisp Object System (Steele Jr. 1990); another example is Cecil (Chambers 1992).

For traditional object-oriented languages, multi-methods represent a useful but optional extension to the basic mechanism. In contrast, in a CIP language, they are essential, since given a constraint `foo(x,y,z)`, we might be determining a value for an unknown `x` using the values for `y` and `z`—in this case dispatching on the type of `x` would be a futile endeavor.

Constraint constructors are special procedures that define the meaning of user-defined constraints. In the simplest case, they are equivalent to rewrite rules: for example, when the constructor `p1=p2` for two Cartesian points `p1` and `p2` is executed, it generates two equality constraints, one for the `x` slots and one for the `y` slots. In general, though, in keeping with Kaleidoscope's imperative nature, constraint constructors may contain not just calls to other constraint constructors, but arbitrary sequences of imperative code. However, all side effects are restricted to the local variables of the constructor. Viewed from the outside, all the constructor is allowed to do is to place other constraints on its arguments. If satisfied, these further constraints will result in the enforcement of the constraint represented by the constructor.

In the remainder of this paper, Section 4.1.2 outlines the evolution of CIP languages. Section 4.1.3 describes how objects and constraints are combined into a unified language model. Constraint constructors are discussed in Section 4.1.4. Section 4.1.5 presents some of the problems in combining constraints and objects and describes a framework that addresses these problems. The Kaleidoscope'93 implementation, the constraint solvers used by Kaleidoscope'93, and future work are discussed in Sections 4.1.6, 4.1.7, and 4.1.8 respectively.

### 4.1.2 Background

A number of experimental programming languages include support for constraints. Much of the activity in this area has been based in logic programming, and includes the CLP and `cc` (concurrent constraint) languages (Cohen 1990; Colmerauer 1990; Jaffar and Lassez 1987; Saraswat 1989; Van Hentenryck 1989; Van Hentenryck, Simonis, and Dincbas 1992; Wilson and Borning 1993). Other constraint languages include Steele's language (Steele Jr. 1980), Bertrand (Leler 1987), and Siri (Horn 1992a; Horn 1992b). (Of these, Siri, another constraint imperative language, is the closest to Kaleidoscope.) For discussions of related work beyond this brief mention, see (Borning, Freeman-Benson, and Wilson 1992; Freeman-Benson and Borning 1992b; Freeman-Benson 1991).

The first version of Kaleidoscope, Kaleidoscope'90, had a Smalltalk-like syntax and served as a proof of concept for CIP. Its successor, Kaleidoscope'91, had several features lacking from Kaleidoscope'90: a conventional Algol-like syntax, multi-methods, and eager constraint solving semantics. These two versions used a refinement model for constraints (as also used in logic programming). In this model, additional constraints further restrict the possible values for variables;

**Table 4.2.** Versions of Kaleidoscope

	<i>Kaleidoscope'90</i>	<i>Kaleidoscope'91</i>	<i>Kaleidoscope'93</i>
<i>Constraint Evaluation</i>	Lazy	Eager	Eager
<i>Variables</i>	Hold streams	Hold streams	Imperative
<i>Concurrent Constraints</i>	Strict	Strict	Non-strict
<i>Syntax</i>	Smalltalk-like	Algol-like	Algol-like
<i>Constraint Model</i>	Refinement	Refinement	Perturbation
<i>Method Dispatching</i>	Single	Multiple	Multiple
<i>Assignment</i>	As a constraint	As a constraint	Destructive

however, variables never change values. Refinement semantics and mutable objects were combined by treating variables as streams of values. Each variable had a stream of states of the variable at different points in time. For example, an assignment statement  $x := x+5$  was treated as a constraint between successive states of  $x$ :  $x_{t+1} = x_t + 5$ . Each time a new constraint was added, a constraint was deleted, or some object changed, all constraints were re-executed finding new values for all slots.

In Kaleidoscope'93 we shifted to a perturbation model. In this model, destructive assignment can change the state of objects (perhaps making previously satisfied constraints unsatisfied), and the system perturbs or adjusts values to reach a new state that best satisfies the constraints. Instead of streams of values, variables in Kaleidoscope'93 refer to a single object, as in conventional imperative languages. Assignment in Kaleidoscope'93 is a destructive state change, accompanied by a once stay constraint on the assigned variable (to prevent the assignment from immediately being undone by some other constraint).

When implemented in its full generality, the refinement model is a more powerful one, since in particular, it doesn't restrict variables to have a single value. However, the perturbation model seems more natural for imperative programmers, and seems to offer more opportunities for improving the efficiency of our implementation. (The refinement and perturbation models are compared at greater length in (Borning, Freeman-Benson, and Wilson 1992).)

Kaleidoscope'91 had a restriction that concurrent constraints execute in nested time scopes since the effects of constructor execution could not be visible until the completion of the constructor (Freeman-Benson 1991). Concurrent constraints allow such things as variable swapping without a temporary (e.g.  $y := x \parallel x := y$ ) but we have relaxed these restrictions in Kaleidoscope'93, making nested time scopes unnecessary, although at some cost in expressiveness.

Table 4.2 summarizes these comparisons between the successive versions of the language.

### 4.1.3 Combining Constraint and Object-Oriented Programming

Kaleidoscope'93 combines constraint and object-oriented programming while preserving a familiar object model from imperative programming. Objects have state and methods, as in most object-oriented languages. Constraints may be placed between objects and object slots, and once a constraint is established, the system attempts to enforce the constraint by filling slots with values. As objects change by assignment, these long-lived constraints re-execute and find new values for their slots.

Adding constraints to an imperative object model adds some complications (Section 4.1.5). Object identity, an important construct in object-oriented languages, has complicated interactions with constraints. The CIP language designer faces a trade-off between expressiveness of the language and problems such as accidental aliasing. Furthermore, circularities can arise if constraints are used to determine classes, values, structure, and identity. (For example, one could imagine a situation in which a given constructor might determine the value of one of its arguments, which determines the class of another argument, which could be the choice of constraint constructor, which could affect the value of the argument . . .) These circularities also need to be addressed in the language semantics. The VICS constraint framework (Section 4.1.5) handles these and other issues resulting from interactions of different constraint types.

There are also positive synergistic effects of combining constraints and objects. For example, constraints that apply to all members of a class can be enforced automatically when a new instance of the class is created. Consider the following:

```
class HorizontalLine subclass of Object;
  var p1, p2;

  initially

new(Point,p1);

new(Point,p2);

always: p1.y = p2.y;
  end initially;
end HorizontalLine;
```

There is a class invariant constraint between the two points representing a `HorizontalLine`. This constraint is enforced automatically at object initialization without the need for an explicit constraint call after initialization and without running the risk that a newly initialized member of `HorizontalLine` does not enforce this invariant.

In the spirit of data abstraction, constraint constructors allow the programmer to define user-defined constraints in terms of other user-defined constraints

or built-in primitive constraints. A user-defined constraint can be used anywhere that a primitive constraint can, and in particular can have a strength in a constraint hierarchy, and a specified duration. This an important property of CIP languages, since it does not restrict the programmer to constraints built into the language.

In the spirit of object-oriented programming, constraint constructors use multi-method lookup rules to select the appropriate procedure or constructor based on the classes of the arguments. For example, a constraint  $j+k=m$  would use the  $+$  constructor for points if  $j$ ,  $k$ , and  $m$  were points, but would use the constructor for arrays if they were arrays.

#### 4.1.4 Constraint Constructors

As described previously, a constraint constructor provides a means of abstraction for constraints. Constraints can be defined in terms of more primitive constraints, similar to a method defining a message in terms of more primitive messages. User-defined constraints may invoke other user-defined constraints, and eventually these constraints are defined in terms of primitive constraints. For example, the following constructor exploits imperative constructs to set up constraints between array cells:

```
constructor +(a, b: Array) = (c: Array);
  local i: Integer;
  for i := 1 to a.size do

always: c[i] = a[i] + b[i];
  end for;
end constructor +;
```

As noted previously, both procedures and constraint constructors use multi-method lookup rules to select the appropriate procedure or constructor based on the classes of the arguments. However, unlike a procedure call, a constraint does not necessarily result in just one execution of the corresponding constructor. The first time a constraint is placed on a set of variables, the multi-method lookup rules are used to find a constructor that implements that constraint, and the selected constructor is executed. If any of the variables change by assignment, then the appropriate constructor is selected using the multiple dispatch rules (perhaps selecting the same constructor as before, or perhaps a different one). The constructor is then executed to enforce the constraint. Dynamically binding constraints to constructors is an important component of the integration of constraints and objects.

#### 4.1.5 VICS Constraint Framework

While designing Kaleidoscope, we found that a considerable number of our language issues dealt with conflicts between identity, constraints, and classes. The

VICS constraint framework provides a framework for handling these different constraint types. (VICS is an acronym for value-identity-class-structure, the four different types of constraint.)

Following Lisp, Kaleidoscope has several different notions of equality: identity, structural equality, and user defined equality. The identity relation between two variables holds if both variables refer to the same object in the computer's memory. Structural equality maintains that two objects are equal if their slots are equal, and two primitive objects are equal by built-in equality primitives. User defined equality allows the programmer to define the conditions for equality between two objects of the same or different classes (e.g. equality between polar and Cartesian points).

An issue in CIP languages concerns when the classes of objects are determined and when objects are initialized, since constructors can execute at arbitrary points in time (unlike procedures whose execution is determined by a procedure call). Should constructors be allowed to initialize objects? Can constructors create objects automatically for uninitialized argument objects? And if so, what is the class of an argument object? Can a constructor output an object that is a member of a subclass of the annotated argument class? Are class annotations constraints themselves? These issues are discussed in the remainder of this section.

Allowing both the class and the slot values of an object to be constrainable led to a whole host of problems resulting from circularities. If a class constraint can depend on a value constraint and the result of the value constraint depends on the result of a class constraint, there can be class/value circularities. More generally, this happens when different types of constraints are allowed to interdepend on each other.

Are structure constraints yet another form of constraint type? For example, the size of an array, or a mirror constraint between two trees, can be considered constraints on structure rather than values. Do we allow these types of constraints with the full generality of value constraints? If so, how does the language overcome circularities?

VICS addresses these issues pertaining to value, identity, class, and structure constraints, factors these different types of constraints, and provides a semantics for avoiding some of the pitfalls relating to conflicts between these constraint types. VICS attempts to strike a balance between expressiveness and flexibility on the one hand, and understandability and ease of solution on the other. The VICS Vapo-Ware Solver<sup>1</sup> is used in the Kaleidoscope'93 implementation to solve VICS constraints (Section 4.1.7). Each of the following subsections describes how the VICS framework handles different constraint types.

---

<sup>1</sup>Since some readers of this chapter will be from countries other than the United States, we provide a brief explanation of the name. Vicks VapoRub is a common household remedy in the U.S. for children's colds. Vaporware is a pejorative term for software that isn't quite real yet. At one point this term unfortunately also applied to our Kaleidoscope solver, but we hope this time has passed.

**4.1.5.1 Value Constraints** In VICS, value constraints are constraints that restrict the contents of an object’s slot to a particular value in its domain. Value constraints are the most common type of constraint discussed in the research literature. (See Sections 4.1.4 and 4.1.7.) Kaleidoscope’93 provides a collection of primitive value constraints in its libraries, which are solved by local propagation.

Kaleidoscope has a powerful approach to value constraints, in which value constraints continue to be enforced as objects change state. Constraints on mutable objects continue to be enforced by re-executing the constructors implementing these constraints, causing more constructors to re-execute, until primitive constraints are handled by the local propagation solver. This contrasts with constraints in the CLP family of languages, where there is no facility for dynamically re-satisfying constraints to deal with state change.

**4.1.5.2 Identity Constraints** Object identity is a fundamental feature in standard object-oriented languages. An important situation in which object identity is visible to the user occurs when two variables  $x$  and  $y$  refer to the same object (i.e.  $x$  and  $y$  are aliased). In such a situation, a state-changing message sent via  $x$  will also be visible when accessing the object referred to by  $y$ . In contrast, if  $x$  and  $y$  refer to equal but not identical objects, changes to the object referred to by  $x$  will not affect  $y$ .

In a constraint imperative language, many of the effects of intentional aliasing can be achieved more cleanly by using persistent equality constraints (e.g. an `always x=y` constraint). However, persistent equality constraints do not handle constructs that in a conventional imperative language would be handled by updating a pointer to refer to a new object. Examples include a pointer that moves down a linked list, splicing a new element into a list, deleting an element from a list, and in-place insertion of a new node in a tree.

As another kind of example, consider the problem of specifying a circular list with constraints. The desired structure is easy to specify in an imperative language that includes pointers. For example, in Common Lisp we could write

```
(setf a (cons 3 nil))
(setf (rest a) a)
```

In Kaleidoscope we might try writing

```
a.head = 3;
a.tail = a;
```

However, if the constraints only imply equality rather than identity, the preceding pair of constraints can be satisfied by an infinite number of different graph structures (with 1, 2, 3, . . . cells forming a circular list, or with a non-circular, infinite list). We might try adding a minimality condition to the Kaleidoscope solver, so that we found the solution with one `cons` cell—but what if we wanted a list of two cells instead?

It would still be possible to avoid object identity by using a different programming style—but one of the design goals for Kaleidoscope was to provide a familiar model for traditional object-oriented programmers, augmented with constraints. This is perhaps the strongest argument for its inclusion.

Based on such considerations, we decided to support a notion of object identity in Kaleidoscope. A simple way of introducing object identity would be to have a `cell` object that could refer to other objects, and that could be re-directed with a `set` message. However, in keeping with the philosophy of the language, we decided instead to introduce identity constraints, in analogy with equality constraints. In addition to the virtue of consistency, using constraints to specify identity may open up some interesting additional programming idioms, for example, describing constraints that keep the back pointers up-to-date in a bidirectional list, or that automatically maintain the threads a threaded tree. Finally, we decided to allow identity constraints to be bi-directional rather than one-way, again in analogy with equality constraints.

In Kaleidoscope'93, an identity constraint is denoted by `==`. Thus, the one-cell list is easily specified by:

```
a.head = 3;  
a.tail == a;
```

However, just as pointers can lead to problems in more conventional languages, identity constraints can lead to problems in CIP languages. The primary problem is accidental aliasing. Identity constraints labeled with an `always` duration (which remain in force throughout the program's execution) don't give rise to these difficulties. However, `always` constraints don't allow such common idioms as having a pointer that marches through a list structure, or a list that can have an element deleted. For such uses, we need `once` identity constraints, just as we have other sorts of constraints with a `once` duration.

The aliasing problem arises when the `once` identity constraint is no longer active. Compare

```
new(Point,a);  
once: a.x = 0;  
once: a.y = 0;  
once: a == b; /* note that this is an identity constraint */  
once: a.x = 5;
```

with

```
new(Point,a);  
once: a.x = 0;  
once: a.y = 0;  
once: a = b; /* note that this is an equality constraint */  
once: a.x = 5;
```

In the first example, the `once` identity constraint still has an effect after its duration, since `a` and `b` remain identical, so that `b.x` is also set to 5. In the

second example, the `once` equality constraint has no effect after it is removed, and `b.x` remains 0.

In Kaleidoscope'93, we have chosen to live with accidental aliasing, despite these problems, since the alternatives we considered led to a less expressive or less intuitive language.

Another issue with object identity concerns circularities. The VICS framework prohibits the constructor for a value constraint from asserting an identity constraint that would invalidate the original choice of constructor for that value constraint. In other words, since the execution of a value constraint cannot depend on an identity constraint executed by that constraint, identity/value circularities are disallowed.

A final issue with object identity is how the identity of uninitialized objects is determined. One approach is to require the programmer to initialize all objects explicitly, and not to allow constructors to create new objects. This is cumbersome, since sometimes we would like constructors to return objects that are created within that constructor. At the other extreme, we considered a very general mechanism (which we called an `identitor`) that would define the identity rules of arguments for all cases of uninitialized arguments. The drawback of this approach is that constructors became verbose, since they need to include code to handle all the different cases for uninitialized arguments, and so the `identitor` mechanism was dropped.

The VICS framework uses neither scheme for handling uninitialized variables. It is important to allow constructors to initialize objects, but `identitors` seemed too powerful, especially considering that most cases for uninitialized arguments will be common cases. The VICS approach is to provide default behavior for uninitialized arguments for these common cases.

Class `Object` is the root class and `LiteObject` is a subclass of `Object`. All uninitialized arguments to a constructor that are annotated as subclasses of `LiteObject` are automatically initialized upon execution of the constructor. Any arguments which are not annotated as subclass of `LiteObject` are considered heavyweight objects (e.g. `Window`). In contrast, these heavyweight objects are not created automatically by the solver; a runtime error is generated if the program tries to invoke an operator on such an uninitialized variable.

As an example, consider the evaluation of `p+q=r`, where `q` and `r` are bound to cartesian points and `p` is unbound. The constructor for `point +` will be invoked:

```
constructor +(a, b: Point) = (c: Point);
  a.x + b.x = c.x;
  a.y + b.y = c.y;
end constructor +;
```

with `p` constrained to be identical to the formal argument `a`, `q` to `b`, and `r` to `c`. Since `Point` is a subclass of `LiteObject`, a new instance of `Point` will be created automatically and bound to `p`.

In contrast, consider `is_selected_window(window1, mouse_position)`. If we evaluate this expression with `window1` unbound, since `Window` is presumably a subclass of `Object` but not `LiteObject`, a new window would (correctly) not be created

automatically by the constructor, and a runtime error would be signaled if the program attempted to perform an operation on it.

**4.1.5.3 Class Constraints** A class constraint is a constraint on the class of a variable. Kaleidoscope'90 treated class constraints in a general way, with class constraints being solved as ordinary constraints. Although this scheme was very powerful, it caused the implementation to be inefficient, was semantically confusing, and gave rise to circularities.

VICS simplifies the semantics and implementation by reducing the power of class constraints. The first design decision was how classes for classless variables are determined, and the second was how constructors are chosen given the classes of variables. (A classless variable is one that is not yet constrained to refer to an instance of a particular class.) Classless variables are considered to be of a special class **Bottom**, until the class is set explicitly by assignment, by declaration, or indirectly through identity constraints. **Bottom** is a subclass of all classes. Thus, a variable of class **Bottom** can be used in multiple dispatch constructor lookup. If there is an ambiguous choice of constructors, then a runtime error is signaled. If there is a single constructor choice, despite one or more classless arguments, then constructor execution can proceed.

For efficiency, VICS currently does not allow backtracking. The order of execution of constraints in an expression is arbitrary. If a later constructor executes and has class information that contradicts an earlier constructor executed in the same expression, a runtime error is generated. Programmers can try to avoid this type of runtime error by making variable classes explicit. Circularities with value and identity constraints are avoided with VICS class semantics. The class of a variable is determined prior to establishing the identity of a variable; value constraints are not allowed to select a class for a variable that would force an alternate choice for the value constraint.

**4.1.5.4 Structure Constraints** Examples of structure constraints include size constraints on arrays, matrix addition (which constrains both the value and structure of matrices), and tree mirror. In some systems these are either treated differently from constraints on values, or not allowed at all. In Kaleidoscope'93, however, constraints on structure are handled in a manner similar to value constraints: either by user-defined constructors, or by primitives (e.g. **Array size**). Structure and value constraints may be intermingled, as long as the solver can solve the structure constraints by local propagation. (This disallows a cycle through a structure constraint, for example, a constraint on the bounds of an array that depends on the contents of an element of that same array.)

## 4.1.6 Implementation

The K-machine is a virtual machine for constraint imperative languages, providing low-level functionality that is unavailable from strictly object-oriented

**Table 4.3.** K-machine Instructions (K-codes)

<i>Operation</i>	<i>Arguments</i>	<i>Description</i>
Branch	BoolVar, NewPC	Conditionally branch to NewPC
CallProc	ProcName, Args	Call a procedure
LoadTemplate	Var, Template	Define a var to refer to a template
AddTemplate	Var	Executes the constraint template
RemoveTemplate	Var	Removes all constraints for the template
Return	none	Returns from a proc. or constructor call
MinStrength	Strgth1, Strgth2	Computes the minimum of two strengths

or strictly constraint-based virtual machines. The interface to the K-machine is similar to that for imperative virtual machines. However, in addition to the standard machine instructions for imperative languages, there are a several instructions pertaining to constraint execution.

An alternative to designing a virtual machine particular to CIP languages would be to use one of the many existing virtual machines for imperative or constraint-based languages. The requirements for CIP support are: object-oriented-support, preferably class-based with inheritance; multi-methods; constraint solving; and dynamically bound constraints. It would be possible, yet extremely awkward, to implement Kaleidoscope with a virtual machine from a conventional object-oriented language with a value-based data store, for example the Smalltalk-80 virtual machine (Goldberg and Robson 1983). However, to do so, the compiler would have to implement the entire semantics of the K-machine in the code generator to ensure that the effect of a constraint-based data store was achieved. This would needlessly complicate the code generator, and would actually reduce the speed of the resulting program. Virtual machines for conventional imperative programming languages are even less suited to CIP languages because they support neither objects nor constraints.

Similarly, it would be possible, yet awkward, to implement Kaleidoscope using the virtual machine for a pure constraint language. For example,  $\text{CLP}(\mathcal{R})$  is a constraint logic programming language whose implementation has a constraint solving engine for constraints over the real numbers (Jaffar, Michaylov, Stuckey, and Yap 1992). The CLAM is the abstract machine used in the  $\text{CLP}(\mathcal{R})$  interpreter, which is based on the WAM, often used in Prolog implementations (Warren 1983). However, to use the CLAM, one would have to translate the Kaleidoscope semantics into one of the object-oriented logic programming schemes. Other constraint-based languages include Bertrand (Leler 1987) and Siri (Horn 1992a). Both Bertrand and Siri are based on an Augmented Term Rewriting virtual machine, which is not powerful enough to support all of the imperative features of Kaleidoscope.

Table 4.3 lists the complete set of K-codes (K-machine instructions). A constraint template sets up a constraint between a set of variables. The execution

of a template causes constructors to execute and primitives to be solved. There are three stacks for constraint templates for each of the three durations: **once**, **always**, and **assert/during**. **Minstrength** is used to compute the minimum of the strength of the currently executing constructor and a second strength (to handle constraints in a constructor that are themselves labeled with a strength).

The key to the semantics of constraint imperative programming is the constraint-based data store and thus, in Kaleidoscope, the object model as well. It is in this constraint-based data store that Kaleidoscope differs from virtual machines for imperative languages. All objects are stored as constraint graph objects. Using the “everything is an object” design principle, stack frames are constraint graph objects as well. Constraints are placed between these constraint graph objects.

The Kaleidoscope’93 implementation is significantly different from the implementation of Kaleidoscope’91. The Kaleidoscope’91 K-machine re-executed all constraints each time a new constraint was added, closely modeling the refinement semantics of the language. The objects as streams of values semantics was mirrored in the implementation by representing variables as streams of objects.

Furthermore, the semantics of Kaleidoscope’91 forced constructors to execute in nested time scopes (Freeman-Benson 1991). Kaleidoscope’91 had an explicit notion of time. In order to prevent the arbitrary advancement of time, the effects within a constructor were local to that constructor via this nested time scope. Handling these nested time scopes led to inefficiencies in the implementation.

In the Kaleidoscope’93 implementation, we have redesigned the K-machine so that constraint constructors execute incrementally. This has improved the performance tremendously, since constructors only need to re-execute when affected by other constraints. Furthermore, objects are no longer represented as streams of values, but persist as mutable objects with a single state, as in conventional imperative virtual machines.

We also eliminated nested time from Kaleidoscope in Kaleidoscope’93, since there is no longer an explicit notion of time. Hence, there are no implementation inefficiencies due to nested time.

#### **4.1.7 Solvers**

The VICS Vapo-Ware Solver is a special-purpose solver used by Kaleidoscope’93 to implement the VICS constraint framework. The VICS framework simplifies class constraints, reducing their expressiveness, but allowing more efficient multi-method lookup rules for solving the classes of classless variables.

Value constraints are handled by constructor execution when a constraint is added. For constraints of long duration, the constraint re-executes each time of any of its variables (or the subparts of those variables) changes identity. The execution of a constructor might trigger additional constructors to execute. This process continues until the remaining constraints are primitive constraints,

which are solved by SkyBlue (Sannella 1993), the incremental local propagation solver used for Kaleidoscope'93 primitives. (SkyBlue is a new algorithm that extends our earlier DeltaBlue algorithm (Freeman-Benson, Maloney, and Borning 1990; Sannella, Maloney, Freeman-Benson, and Borning 1993) to allow constraints with multiple outputs, and that provides better handling of cycles.)

Structure constraints, as described in Section 4.1.5, are grouped semantically with value constraints, and thus share the same implementation. Hence, there are primitives for structure constraints as well as value constraints.

Identity constraints are solved by special-purpose constructors that establish an identity relation between two variables. For identity constraints of long duration, this constructor is re-executed each time one of these variables changes identity. An unbound variable that is annotated with a subclass of `LiteObject` is bound at constructor execution time, as specified by the VICS semantics. Finally, there is a predicate that tests whether two variables are identical, also implemented as a constructor.

All primitive constraints are represented as constraint edges in the primitive constraint graph. SkyBlue solves these constraints by executing the primitive constraints and finding values for all variables. As new primitive constraints are added to this graph as a result of constructor execution, SkyBlue re-executes primitives to satisfy all primitive constraints. Since SkyBlue is incremental, constraints that are not affected by changes to the primitive graph do not re-execute.

#### 4.1.8 Future Work

As of this writing, our Kaleidoscope'93 implementation has just become usable. We plan to continue work on the implementation, and as it stabilizes, begin to write larger programs in the language, and feed back the results to the language design and implementation. Another major effort will involve increasing the efficiency of the code produced by the Kaleidoscope compiler, in particular to eliminate runtime constraint satisfaction when possible. Finally, we are about to replace our primitive constraint solver, SkyBlue, with a more powerful solver, CobaltBlue. In addition to local propagation constraints, CobaltBlue will handle simultaneous equations and non-unique constraints.

#### Acknowledgements

Many people have given help and advice on this work; we would like to thank in particular Craig Chambers, Denise Draper, Jens Palsberg, Michael Sannella, and Michael Schwartzbach. This work was supported in part by the National Science Foundation under Grant No. IRI-9102938, by the Canadian National Science and Engineering Research Council under Grant OGP0121431, by a Fellowship from Apple Computer for Gus Lopez, and by equipment grants from Sun Microsystems.



## Bibliography

- Borning, A., B. Freeman-Benson, and M. Wilson (1992, September). Constraint hierarchies. *Lisp and Symbolic Computation* 5(3), 223–270.
- Chambers, C. (1992, June). Object-Oriented Multi-Methods in Cecil. In *Proceedings of the 1992 European Conference on Object-Oriented Programming*, pp. 33–56.
- Cohen, J. (1990, July). Constraint logic programming languages. *Communications of the ACM* 33(7), 52–68.
- Colmerauer, A. (1990, July). An introduction to Prolog III. *Communications of the ACM*, 69–90.
- Freeman-Benson, B. (1991, July). *Constraint Imperative Programming*. Ph.D. thesis, University of Washington, Department of Computer Science and Engineering. Published as Department of Computer Science and Engineering Technical Report 91-07-02.
- Freeman-Benson, B. and A. Borning (1992a, April). The design and implementation of Kaleidoscope'90, a constraint imperative programming language. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, pp. 174–180.
- Freeman-Benson, B. and A. Borning (1992b, June). Integrating constraints with an object-oriented language. In *Proceedings of the 1992 European Conference on Object-Oriented Programming*, pp. 268–286.
- Freeman-Benson, B., J. Maloney, and A. Borning (1990, January). An incremental constraint solver. *Communications of the ACM* 33(1), 54–63.
- Goldberg, A. and D. Robson (1983). *Smalltalk-80: The Language and its Implementation*. Addison-Wesley.
- Horn, B. (1992a, October). Constraint patterns as a basis for object-oriented constraint programming. In *Proceedings of the 1992 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, British Columbia, pp. 218–233.
- Horn, B. (1992b). Properties of user interface systems and the Siri programming language. In B. Myers (Ed.), *Languages for Developing User Interfaces*, pp. 211–236. Boston: Jones and Bartlett.

- Jaffar, J. and J.-L. Lassez (1987, January). Constraint logic programming. In *Proceedings of the Fourteenth ACM Principles of Programming Languages Conference*, Munich.
- Jaffar, J., S. Michaylov, P. Stuckey, and R. Yap (1992, July). The CLP( $\mathcal{R}$ ) language and system. *ACM Transactions on Programming Languages and Systems* 14(3), 339–395.
- Leler, W. (1987). *Constraint Programming Languages*. Addison-Wesley.
- Sannella, M. (1993, February). The SkyBlue Constraint Solver. Technical Report 92-07-02, Department of Computer Science and Engineering, University of Washington.
- Sannella, M., J. Maloney, B. Freeman-Benson, and A. Borning (1993, May). Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm. *Software—Practice and Experience* 23(5), 529–566.
- Saraswat, V. A. (1989, January). *Concurrent Constraint Programming Languages*. Ph. D. thesis, Carnegie-Mellon University, Computer Science Department.
- Steele Jr., G. L. (1980, August). *The Definition and Implementation of a Computer Programming Language Based on Constraints*. Ph. D. thesis, MIT. Published as MIT-AI TR 595, August 1980.
- Steele Jr., G. L. (1990). *Common Lisp: The Language* (second ed.). Bedford, Massachusetts: Digital Press.
- Van Hentenryck, P. (1989). *Constraint Satisfaction in Logic Programming*. Cambridge, MA: MIT Press.
- Van Hentenryck, P., H. Simonis, and M. Dincbas (1992, December). Constraint satisfaction using constraint logic programming. *Artificial Intelligence* 58(1–3), 113–159.
- Warren, D. H. D. (1983, October). An abstract prolog instruction set. Technical Report 309, SRI International, Menlo Park, California.
- Wilson, M. and A. Borning (1993, July, August). Hierarchical constraint logic programming. *Journal of Logic Programming* 16(3 & 4), 277–318.