Implementing Constraint Imperative Programming Languages:

The Kaleidoscope'93 Virtual Machine

Gus Lopez, Bjorn Freeman-Benson[1], and Alan Borning
Department of Computer Science and Engineering
University of Washington
Seattle, Washington 98195

July 1994

Technical Report 94-07-07

[1]Carleton University, School of Computer Science,
514 Herzberg Building, 1125 Colonel By Drive, Ottawa, Ontario, Canada, K1S 0G9, bnfb@scs.carleton.ca

Abstract. Constraint Imperative Programming (CIP) languages integrate declarative constraints with imperative state and destructive assignment, yielding a powerful new programming paradigm. However, CIP languages are difficult to implement efficiently due to complex interactions between the two donor paradigms. Neither the virtual machines for classical object-oriented languages, nor those for existing constraint languages, are suitable for implementing CIP languages, as each assumes a purely imperative or a purely declarative computation model. We have developed a new virtual machine for CIP languages, the K-machine, an imperative machine with an incremental constraint solver and a constraint-based, rather than value-based, data store. This virtual machine allows user-defined constraints to be defined using constraint constructor definitions which are the CIP analog to method definitions. Similar to methods, these constructors are able to reference variables indirectly through many levels of pointers. The K-machine maintains relations between objects in the presence of state change to these indirectly referenced objects. The K-machine is capable of supporting a wide variety of CIP languages, including our most recent: Kaleidoscope'93.

To appear in Proceedings of OOPSLA'94
Portland, Oregon, October 1994

Implementing Constraint Imperative Programming Languages: The Kaleidoscope'93 Virtual Machine

Gus Lopez, University of Washington
Bjorn Freeman-Benson, Carleton University
Alan Borning, University of Washington

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

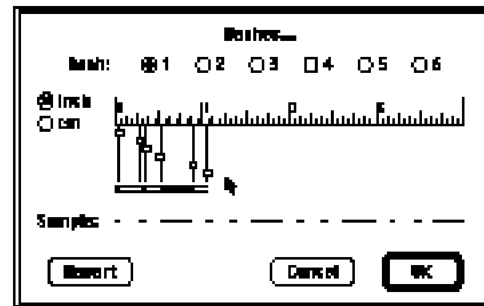
Bjorn Freeman-Benson
School of Computer Science, Carleton University
1125 Colonel By Drive
Ottawa, Ontario, Canada K1S 5B6
bnfb@scs.carleton.ca

Abstract

Constraint Imperative Programming (CIP) languages integrate declarative constraints with imperative state and destructive assignment, yielding a powerful new programming paradigm. However, CIP languages are difficult to implement efficiently due to complex interactions between the two donor paradigms. Neither the virtual machines for classical object-oriented languages, nor those for existing constraint languages, are suitable for implementing CIP languages, as each assumes a purely imperative or a purely declarative computation model. We have developed a new virtual machine for CIP languages, the K-machine, an imperative machine with an incremental constraint solver and a constraint-based, rather than value-based, data store. This virtual machine allows user-defined constraints to be defined using constraint constructor definitions which are the CIP analog to method definitions. Similar to methods, these constructors are able to reference variables indirectly through many levels of pointers. The K-machine maintains relations between objects in the presence of state change to these indirectly referenced objects. The K-machine is capable of supporting a wide variety of CIP languages, including our most recent: Kaleidoscope'93.

Keywords

constraints, constraint imperative programming, incremental constraint solving, virtual machines



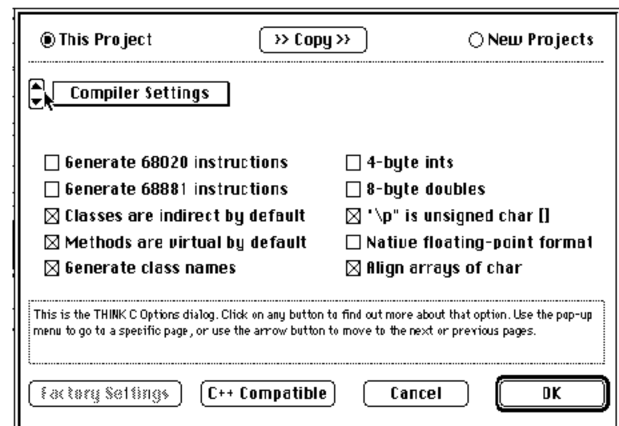


Figure 1. Example User Interfaces

1. Introduction

Imperative programming languages are well understood, used by a large number of programmers, and well supported by software tools. However, even object-oriented imperative languages are often lower level than one would like. Consider graphical user interfaces such as those that initially motivated us, e.g., the MacDraw dashed-lines dialog box in Figure 1 (top), the Think C options window in Figure 1 (bottom), and the thermometer in Figure 3. We observed

Imperative

```
while mouse.button = down do
    old ← mercury.top;
    mercury.top ← mouse.location.y;
    temperature ← mercury.height / scale;
    if old < mercury.top then
        paint_rect( grey, mercury.top, old );
        display_number( temperature );
    elseif old > mercury.top then
        paint_rect( white, mercury.top, old );
        display_number( temperature );
    end if;
end while;
```

Constraint Imperative

```
always: temperature = mercury.height / scale;          (1)
always: white_rectangle.top = thermometer.top;         (2)
always: white_rectangle.bottom = mercury.top;          (3)
always: mercury.bottom = thermometer.bottom;           (4)
always: color(white_rectangle,white);                  (5)
always: color(mercury,grey);                           (6)
always: display_number( temperature );                 (7)
while mouse.button = down assert
    mercury.top = mouse.location.y;                    (8)
end while;
```

Figure 2. Imperative Code versus CIP Code

that some portions of these interfaces are most clearly and conveniently described using constraints—automatically maintained relations between variables—while other portions are most clearly described using standard imperative constructs such as assignments and sequencing. However, no existing language used to program these interfaces directly supported both constructs. Thus, although these user interfaces may be written in high-level imperative languages, the constraint portions of the user interface are written at a low level, by hand, and enforced by a code fragments distributed throughout the program—a recipe for maintenance headaches. To address this problem, we proposed, in [Freeman-Benson 91] and [Freeman-Benson & Borning 92], an integration of two disparate paradigms: a standard object-oriented imperative one, and a declarative constraint one. The result is named constraint imperative programming (CIP). While the original motivation for CIP languages was interactive applications, CIP languages are actually general-purpose languages—in fact, they are a superset of traditional object-oriented languages.

Consider a slider widget that allows the user to drag the mercury of a thermometer up and down with the mouse (Figure 3), and two code fragments (Figure 2) for achieving this effect. The version in Figure 2 (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 imperative version in Figure 2 (right) uses of a combination of imperative constructs and

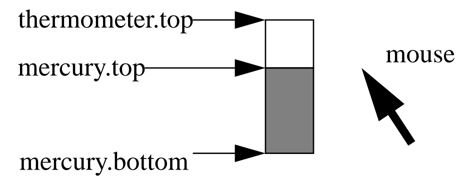


Figure 3. Thermometer

constraints. Some of the constraints specify relations that must always hold, e.g., lines 1–7, while others specify relations that should hold only while a given condition is true, e.g., line 8. Imperative constructs, such as the `while` statement, are used to control program execution (in particular, when certain constraints should hold). The constraint imperative version on the right is both higher-level and more maintainable than the imperative version on the left.

The constraints used in constraint imperative languages are declarative statements of relations among elements of the language's computational domain, e.g., integers, booleans, strings, and other objects. These constraints solved by the language's embedded constraint solver, and their usefulness stems from the fact that they emphasize the relation rather than the procedural steps necessary to maintain that relation. CIP languages are general purpose programming languages; they implement general purpose multi-directional constraints, rather than a uni-directional, or dataflow, subset. For an overview of constraints and constraint programing, see [Freeman-Benson et al. 90] or [Leler 87].

The three fundamental problems of CIP language implementation that our K-machine is designed to solve are as follows.

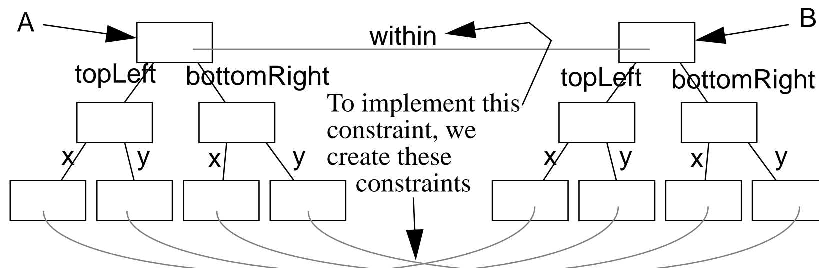


Figure 4. Non-renewable constraint

i. The imperative and constraint paradigms conflict on the issue of control of variable values: the imperative paradigm gives the precise control of each variable's value to the programmer, while the constraint paradigm gives the responsibility to the constraint solver. In an imperative program, an assignment is the only way to update the value of a variable, and each assignment only updates one variable[1]. In a constraint program there are no assignments, so variable values are modified by adding and removing constraints, yet with each addition or removal the constraint solver may change any number of variable values so as to satisfy the remaining constraints. Thus any integration of the imperative and constraint paradigms must include communication between the two. In the K-machine, this problem is handled by replacing the value-based store of an imperative machine with a constraint-based store. The imperative engine notifies the constraint solver whenever a constraint is to be added or deleted, and the constraint solver handles queries from the imperative engine for the values of variables.

ii. A CIP language will obviously have a mechanism for creating constraints over the built-in primitive domains (integers, booleans, real numbers, etc.). An object-oriented CIP language must also have a mechanism for defining and creating constraints over complex user-defined objects. For example, the programmer who is coding the dialog box in Figure 1 (top) should constrain lines, rectangles, dash components, check boxes, etc. rather than the integer and real number components of those objects. Thus, the programmer should write left_of(dash[i],dash[i+1]) rather than violating object encapsulation to write constraints of the form dash[1].dialog_rectangle.bottom_right.x < dash[2].dialog_rectangle.top_left.x, etc. Constraints of this latter form are not object-oriented because (a) they violate public-private encapsulation boundary of instance variables, and (b) they prevent the programmer from using a different implementation of the same abstraction, e.g., a top/bottom/left/right implementation of rectangles instead of a topLeft point/bottomRight point implementation. (This problem is also discussed in [Freeman-Benson & Borning 92].)

iii. If the part-whole structure of the objects were guaranteed not to change, then user-defined constraints on user-defined objects could be implemented using properly encapsulated methods that recursively descend the structure and create primitive constraints on the leaves. For example, consider the within constraint between two rectangles (A and B) illustrated in Figure 4. One could implement this using a within method in class Rectangle which calls the aboveRight and belowLeft methods in class Point, which call the + and - methods in class Integer which, finally, create primitive constraints between the x and y components of the points. No special mechanisms, other than primitive constraints, are necessary.

However, the effect of using methods is that the within(A,B) constraint is implicit rather than explicit—there is no longer an explicit representation of the within constraint; instead, it is represented implicitly by the primitive constraints on the subparts of A and B. Thus, if a new object were assigned to A, then the implicit constraint

1. Ignoring the effect of aliasing. As an aside, we note that constraints provide a more general and disciplined mechanism than that provided by aliasing.

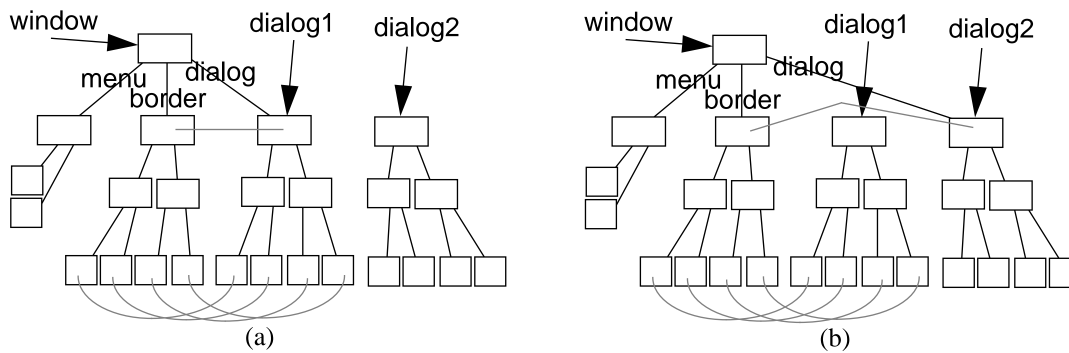


Figure 5. Incorrect constraints after structure change

would become within(A_{former_value},B) rather than within($A_{current_value}$,B). In other words, the new rectangle stored in A would not be constrained to stay within B, much to the surprise of the programmer who had expected the within(A,B) message to ensure that the A rectangle always be contained within the B rectangle.

A real-life example of this situation is the dialog box window in Figure 1 (bottom): the window can show any one of four or five dialogs, depending on the item selected from the menu in the upper left. The object structure for this window is shown in Figure 5. In Figure 5(a), dialog box one is selected and the dialog instance variable of the window object contains to the dialog1 object. The desired constraint, within(border,dialog), represented by the dashed line has been implemented by creating primitive constraints (the gray lines) between the x and y variables of the component points. When the user selects dialog box two in Figure 5(b), the dialog2 object is assigned to the dialog instance variable and thus the part-whole structure of the window is changed. Unfortunately, the desired constraint, within(border,dialog), again represented by the dashed line, is no longer correctly implemented! In fact, the old dialog1 object remains visible rather than the selected dialog2 object. Obviously using primitive constraints to implement complex constraints implicitly is inadequate, and a more powerful explicit constraint mechanism is necessary.

A general purpose CIP language must have a mechanism for maintaining constraints on user-defined objects even when components of those objects change identity. Most of the machinery in the K-machine exists to provide exactly such a mechanism.

In the remainder of this paper, we discuss implementation considerations common to all constraint imperative programming languages, as well as our implementation of Kaleidoscope'93 [Lopez et al. 93], hereafter referred to as Kaleidoscope. The K-machine is a general CIP language virtual machine, and is not restricted to the particular choice of primitive domains, constraint solvers, and inheritance model used in Kaleidoscope. This implementation demonstrates that the constraint and imperative paradigms can be integrated at the virtual machine level, and accessed via an interface similar to that for imperative virtual machines. Section 2 presents an overview of the Kaleidoscope'93 language and our current implementation. Section 3 outlines the K-machine, a virtual machine for CIP languages with an incremental constraint solver, and Section 4 describes the constraint-based data store, which augments a conventional imperative data store with constraints linking values. Related work is discussed in Section 5, and conclusions and future work are presented in Section 6.

2. The Kaleidoscope'93 Language

Kaleidoscope'93 is similar to many other object-oriented languages: it has classes, objects with mutable state, methods, destructive assignment, and so forth. An object's state can be changed by sending messages to it. The key difference between constraint imperative programming and imperative programming is the ability to relate variables (such as slots/instance variables,

locals, globals, etc.) by constraints. When a variable has one or more constraints on it, the constraint solver is allowed to alter the binding of the variable, or the state of the object bound to the variable, to satisfy the constraints.

Just as a method definition extends an imperative language with user-defined messages, a constraint constructor definition extends a CIP language with user-defined constraints. However, in contrast to methods, constructors need to be re-evaluated when the constrained objects change, since constraints might no longer be satisfied as a result of these changes. The most flexible constraint model would allow constraints to be asserted and retracted at arbitrary points in time. Although the K-machine is capable of supporting this model, we felt its use at the language level could lead to difficulties in predicting behavior, since any piece of code could alter the active set of constraints. Instead, we adopt a structured design for Kaleidoscope'93, in which the static program text determines when constraints are active. (We might relate this to the GOTO statement/structured programming controversy of the 1960's: constructs that allow constraints to be asserted and retracted at arbitrary times are analogous to GOTO statements, while the control structures in Kaleidoscope are analogous to structured control statements.)

The default constraint duration is always, which causes a constraint to remain active for the duration of the program. For example, if we would like the cursor to follow mouse movements, this can be achieved with an always constraint:

```
always: mouse.position = cursor.position;
```

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. For instance, when an application starts up, the initial position of the cursor might be the center of the screen. However, that position should be unconstrained thereafter so that subsequent mouse movements allow the cursor to move:

```
once: cursor.position = screen.center;
```

In this example, the = constraint is enforced between cursor.position and screen.center, and is then retracted, leaving the effects of constraint satisfaction. Assignment statements are a particular kind of once constraint, in which the value of the expression on the right hand side is determined at one instant, then at the next instant a one-way once constraint is applied between this value and the expression on the left. This mechanism integrates assignment with the constraint system, and at the same time allows such standard assignments as x := x+5. (A once: x = x+5 constraint would be unsatisfiable.) The alternative of making assignment statements into conventional load and store sequences would be possible as well, but would complicate the semantics and would still require tight communication between the imperative and constraint engines (Problem i listed in Section 1).

Finally, the during construct specifies that a constraint should remain in force during the execution of a block or loop. The following example asserts that the window position and mouse position are the same while the mouse button is down:

```
assert mouse.position = window.position
    during
        while mouse.button = down do
        ....
        end while;
```

We have found it useful to extend the constraint paradigm to allow both *required* and *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). These hierarchies are useful in determining a programmer's preferences when a system of constraints is under-constrained or over-constrained. Further information on constraint hierarchies can be found in [Borning et al. 92].

Due to its object-oriented nature, constraints in Kaleidoscope are considerably different from constraints in other language families. Since languages in the Constraint Logic Programming family do not provide a facility for objects with mutable state, there is no automatic mechanism for re-satisfying a constraint as a result of a state change. (See references [Cohen 90], [Colmerauer 90], [Jaffar & Lassez 87], [Van Hentenryck 89], [Van Hentenryck et al. 92], and [Wilson & Borning 93].) Other CIP languages do not allow con-

straints between arbitrary objects, and restrict constraints to instance variables. For instance, Siri, another CIP language that is probably the closest relative to Kaleidoscope'93, only resatisfies constraints between instance variables within the representation of a single object [Horn 92a].

Most constraint languages restrict constraints to those than can be expressed over built-in primitive domains. As mentioned earlier, limiting constraints to primitive domains would be overly restrictive in an object-oriented language, since user-defined domains (i.e. classes) are frequently used in object-oriented programs. One of Kaleidoscope's novel features is the concept of a *constraint constructor*, which allows a constraint to be defined in terms of more primitive constraints. Constructors allow definitions of user-definable constraints, similar to the way methods are implemented in terms of more primitive message sends. Eventually, all user-defined constraints reduce to primitive constraints, which are handled by the solvers over these built-in primitive domains.

The astute reader will notice that if the object structure is not allowed to change, then constraint constructors are identical to normal procedures and methods that create constraints. However, in realistic object-oriented programs, such as the windows in Figure 1 (implemented as shown in Figure 5), the part-whole structure of some objects does change, and thus constraint constructors are not just methods. The fundamental difference is that procedures or methods execute only as a result of an explicit procedure call or message send, whereas constructors re-execute automatically as a result of state changes in their constrained objects and variables. Constructor execution semantics are not as straightforward as procedure execution, since the language implementation needs to determine which constraints are affected by any change, which constraints need to be re-satisfied by constructor calls, and which other variables need to change as a result. In our scheme for implementing CIP languages, the bookkeeping required for maintaining constructor execution semantics is handled at the virtual machine level, significantly simplifying code generation.

Constructors are dynamically dispatched using multi-method lookup, in which all the arguments are significant in selecting the constructor, rather than the more traditional single dispatching. Multi-methods are used in a number of other object-oriented languages, for example CLOS [Steele Jr. 90] and Cecil [Chambers 92]. In Kaleidoscope, multi-methods are essential, since for some constructor calls, the first argument might be unbound. For example, we might call +(x,y,z), with y and z bound to Vectors and x to be determined by the constraint solver, in which case the + constraint might be bound to the +(Vector, Vector, Vector) constructor and x would become a Vector.

2.1 Implementation Overview

The Kaleidoscope'93 implementation consists of a compiler, a primitive constraint solver, and a specialized virtual machine, the K-machine. All three are currently implemented using the Common Lisp Object System. A Kaleidoscope program is a collection of class, procedure, and constraint constructor definitions, and a single initial procedure call. The compiler translates Kaleidoscope programs into K-machine instructions, a.k.a. K-codes. The K-machine is derived from an imperative virtual machine, and contains a code store, data store, stack, program counter, and various general purpose registers. Figure 6 illustrates the components of the Kaleidoscope'93 language implementation.

Our compiler performs some optimizations to reduce or eliminate expensive run-time constraint solving, such as inlining to avoid constraint satisfaction in cases where an object's class can be statically determined as primitive, as well as a few standard compiler optimizations such as constant folding, code motion, and dead-code elimination. However, in contrast to imperative languages, the chief bottleneck to CIP languages is constraint solving, and so a major focus of our future work will be investigating additional optimizations to reduce or eliminate run-time constraint solving (Section 6).

3. The K-machine

The K-machine interprets K-code instructions. These instructions include typical imperative instructions,

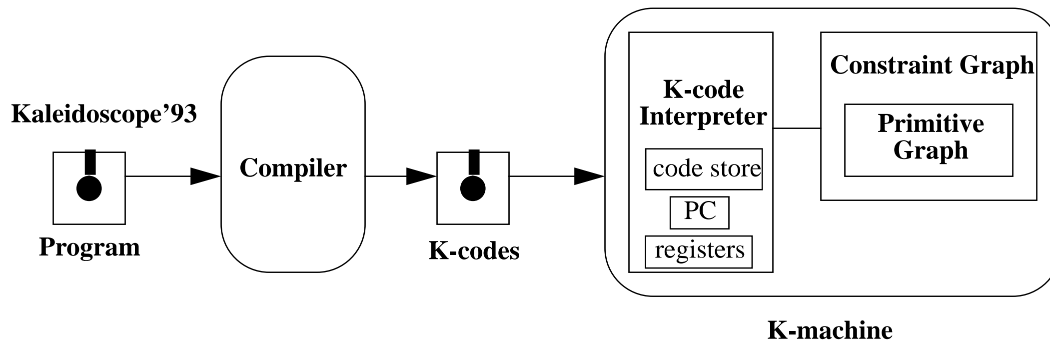


Figure 6. Kaleidoscope'93 Implementation

such as Add, Load, and Branch, as well as the specialized instructions listed in Figure 7 and discussed below. The K-machine is derived from imperative virtual machines and contains many of the same components. It differs from standard imperative VMs in that it supports constraint-based operations on objects. The value-based data store of imperative machines is extended into a constraint-based data store by allowing objects to be connected by relations.

In the purest sense, a constraint-based data store is a set of constraints which, when taken together, determine values for variables. (This is the approach used in the cc family of languages [Saraswat 93].) Variables in this case are as in mathematics, rather than naming changeable storage locations. However, for efficiency reasons, the K-machine data store does contain traditional imperative variables, as well as the constraints that determine their values. The data store is represented as a graph whose nodes represent variables and edges represent both pointers and constraints (i.e., there are two types of edges: one type represents traditional pointers from variables to objects, and the other type represent constraints between variables). The constraint edges are either constructed or primitive. Constructed constraint edges remember the child constraints that they have created. Primitive constraint edges can only be placed between variables (nodes) containing objects from the built-in domains (which are numbers, characters, strings, and booleans in Kaleidoscope'93). In order to integrate constraints with objects, additional bookkeeping is required to maintain the constraints as the objects change state. After each state change, the system identifies constraints linking the changed variable to other variables and objects, finds constructors to carry out these changes, executes code to resatisfy those constraints, finds further changed variables, and so forth.

At the K-machine level, constructors and procedures have identical representations: a signature and a block of K-codes. To ensure that constructors behave as relations, the Kaleidoscope language definition requires that all side effects be restricted to the local variables of the constructor. Viewed from the outside, all the constructor is allowed to do is place other constraints on its arguments. If satisfied, these further constraints will result in the enforcement of the higher-level constraint represented by the constructor. Although constructors and procedures have identical representations, the K-machine handles their executions differently. Procedure calls are handled as in a traditional object-oriented language: the CallProc K-code selects and invokes the appropriate procedure using multi-method lookup. Constraints, however, are handled by *constraint templates*. A constraint template is, essentially, a continuously repeating procedure call except that it calls a constraint constructor rather than a procedure. A template is created for each instance of a constraint and keeps track of the variables being constrained and the name of the constraint being applied. For an always constraint, a constraint template is simply asserted using AddTemplate. For a once constraint, the template is asserted and immediately retracted using an AddTemplate, RemoveTemplate pair. Finally, for an assert/during constraint, the template is added prior to, and removed following, the block of code.

A template is executed once it is added, dynamically bound to a constraint constructor, and invoked. Logically, we can view templates as all being re-executed

Operation	Arguments	Description
CallProc	ProcedureName, Arguments	Call a procedure
Return		Return from a procedure or constructor
MinStrength	Strength, Result	Minimum of the current strength and Strength
LoadTemplate	Var, Template	Define a Var to refer to a constraint template
AddTemplate	Var	Add and Execute the constraint template
RemoveTemplate	Var	Remove all constraints for the constraint template
TrueBranch	Condition, NewPC	Branch to NewPC if Condition is True
FalseBranch	Condition, NewPC	Branch to NewPC if Condition if False
ClassBranch	Var1, Var2, NewPC	Branch to NewPC if Var1 and Var2 are members of the same class
PrimitiveAssignment	Var1, Var2	Optimized assignment for case where classes of Var1 and Var2 are primitive
Unbind	Var	Clears the variable
New	Var, Class	Var is initialized with an object of class Class
PrimitiveAction	Number, Args	Built-in unnamed operation

Figure 7. K-machine Instructions (K-codes)

following each state change, so as to maintain user-defined constraints. In fact, the Kaleidoscope'91 implementation did exactly that [Freeman-Benson 91]. However, while this made constraint maintenance straightforward, most of these constructor re-executions were superfluous as, in the vast majority of cases, they filled slots with exactly the same values as the slots had previously held. Further, although the Kaleidoscope'91 primitive solvers were incremental, the continual re-execution of constructors and regeneration of primitive constraints did not allow the implementation to exploit these incremental algorithms.

The K-machine avoids this bottleneck by re-executing constraint templates only when they might be affected by a state change. This interacts with the changing object structure problem in two ways. Changing an object's structure changes the components that need to be constrained. Furthermore, changing an object's structure might change which constructors are invoked to satisfy its constraints. Incrementally satisfying constraints requires additional bookkeeping in the K-machine, but the cost is small compared to the tremendous performance advantage of incremental constraint satisfaction over repeatedly solving all constraints. Further, this design allows us to exploit the incremental properties of our local propagation solver [Freeman-Benson et al. 90], [Sannella 93]. See Section 4.1 for more details.

To support the incremental execution of constraints, each variable maintains two lists: the *upstream variables*, which were used to compute that variable's current value, and the *downstream templates*, which are all templates whose choice of, or execution of, a constructor could possibly depend on that variable's value. Assignments and computational primitives update the upstream variables list, and constraint constructors use this list to update the downstream templates list. For example, after executing the trivial program:

```
always: A + E = F;
A := B + C; D := A * G;
```

A's upstream variables list is {B,C}, and D's upstream variables list is {A,B,C,G}. A's downstream templates list is {+}, and D's downstream templates list is {}.

Incremental constraint satisfaction is triggered by assignment. When a variable is assigned to, all the constraint constructors depending on that variable (i.e., in the second list) are re-executed. A constraint is re-executed by first removing the constraint edges placed on component objects by the constructor, and then executing the code for the selected constructor. This avoids having to re-execute constraints when a component changes that is unaffected by the constructor.

To illustrate the incremental execution of constructors, consider the Kaleidoscope program in Figure 8. The constructor +(Point, Point, Point) is chosen at line (1)

since p1 and p2 contain objects of class Point. The Point object in p3 becomes the sum of p1 and p2. The + constructor is not re-executed after the assignments to p4.x and p4.y at line (2) because these state changes have no effect on p1, p2, and p3 themselves, but only on their components. The assignment to p2 in (3) however, requires that the p1 + p2 = p3 constraint be re-satisfied, since the constraint no longer holds as a result of the assignment to p2. The appropriate constructor is chosen (again, the +(Point, Point, Point) constructor) and executed.

An assignment v := expr may change the identity of the object to which the variable v refers, which may necessitate using different constructors to satisfy the constraints attached to v. This automatic and incremental re-satisfaction of constraints after an assignment is the solution to the "changing object structure" problem illustrated in Figure 5 — thus this problem does not occur in CIP languages implemented using the K-machine.

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

```
procedure start ()
    var p1, p2, p3, p4: Point;
    p1 := 2@2; p2 := 10@10;
    always: required p1 + p2 = p3;          (1)
    p4.x := 100; p4.y := 100;               (2)
    p2 := p4;                               (3)
end procedure start;
start;
```

Figure 8. Incremental constraint re-satisfaction

4. Constraint-Based Data Store

In the Smalltalk tradition, Kaleidoscope language and implementation components are represented as objects, including all user-defined objects, primitives such as numbers and booleans, and even system objects such as stack frames. Kaleidoscope objects are stored in a constraint graph consisting of nodes, objects, and constraint edges. This uniform treatment simplifies the K-machine implementation. If there are no constraint edges between objects, then the constraint store resembles a conventional imperative store. Thus, one can consider a constraint store as a generalization of an

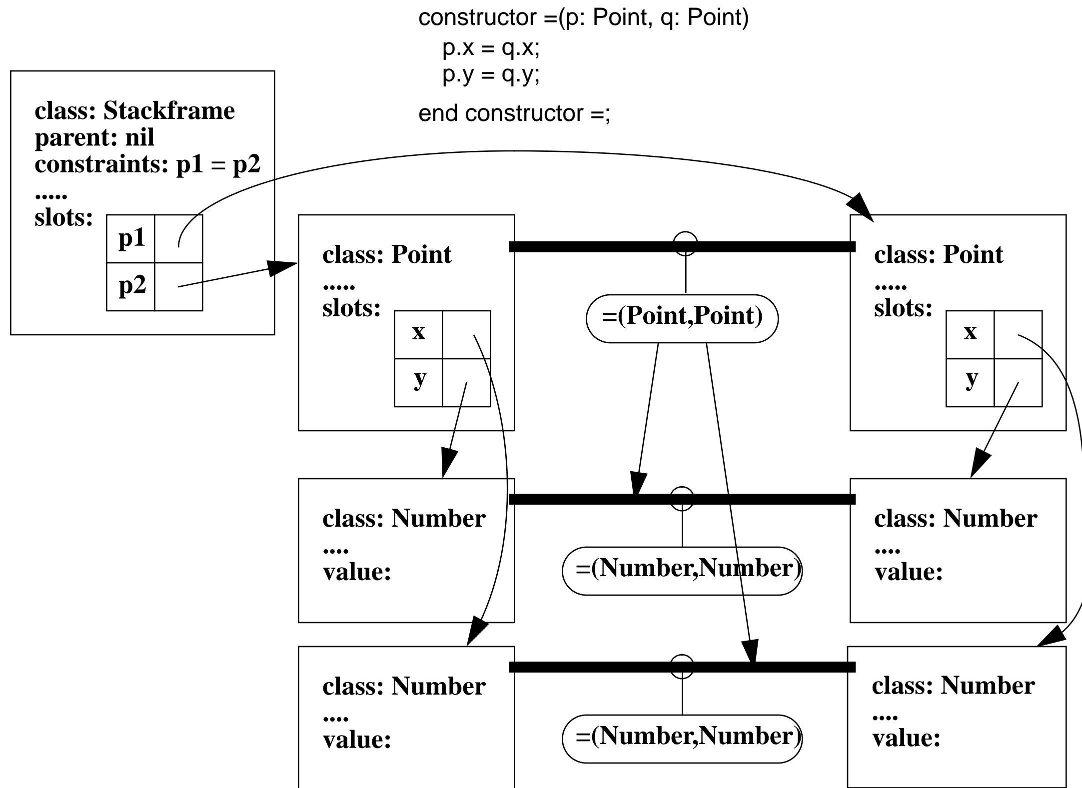


Figure 9. Constraint Graph Representation of Objects

imperative store where variable values may also be determined by constraints in addition to identity assignment.

Figure 9 illustrates the representation of an equality constraint between two points. The large boxes are objects, the small labelled boxes which point to objects are variables, and heavy lines indicate constraints (labelled by ovals with the selected constructor). When the = constraint template is added, a constraint edge is placed between p1 and p2. Multi-method lookup rules are used to find a constructor with the same name and arity as this constraint edge. The =(Point, Point) constructor executes and places two new constraint edges on the graph, one between the x slots of p1 and p2 and one between the y slots of p1 and p2. Continuing in the same fashion, the = constraints between the x and y slots are satisfied by finding constructors, =(Number, Number). These constructors place primitive constraint edges on the constraint graph. User-defined constraints eventually bottom out at primitive constraints over built-in domains. The primitive constraint solver uses local propagation and Gaussian elimination to complete the computation.

The Kaleidoscope constraint store is implemented entirely in Common Lisp. Garbage collection is handled by Lisp's garbage collection since the Kaleidoscope constraint store eliminates all references to objects that are inaccessible at the Kaleidoscope language level.

4.1 Primitive Constraint Solvers

Kaleidoscope'93 uses two different types of primitive constraint solvers, one for finding variable values in particular domains (e.g., booleans, numbers, and strings) and one for determining object identity and structure. The former is the familiar notion of constraint solver from constraint programming languages which satisfies constraints over particular value domains and the latter is used to solve for object identity.

Most constraint languages solve constraints over different value domains, such as booleans, numbers, colors, and strings. We term these constraints *value constraints*, since the satisfaction of these constraints finds values for objects within their domains. The primitive constraint solver we currently use in the Kaleidoscope'93 implementation is CobaltBlue. CobaltBlue is an extension of SkyBlue [Sannella 93], and can solve simultaneous equations, multiple output and non-unique constraints incrementally by local propagation. However, the K-machine design is general enough to accommodate other solvers.

CIP languages combine imperative constructs such as object identity and class membership with declarative constraints. Naturally, these constructs can also be specified by constraints. One of the fundamental concepts of object-oriented programming, object identity, can result in implicit relations, even when explicit identity constraints are supported. Furthermore, by allowing constraints on object identity, object structure, class membership, and object value, complex interactions, such as circularities, can occur between any two different categories of constraints. To deal with these interactions, we developed the VICS (value/identity/class/structure) framework, to factor out conflicts between object value and object identity.

Identity constraints are used to treat object identity as a declarative relation that is compatible with the Kaleidoscope constraint model. The VICS Vapo-Ware solver is used by the K-machine to categorize constraints (by value and identity), and distribute them to the appropriate sub-solver. Similar to value constraints, identity constraints are solved by local propagation, however the satisfaction of these identity constraints determines variable references instead of object values. Identity constraints and the VICS Vapo-Ware solver are discussed in [Lopez et al. 94].

5. Related Work

The special implementation needs of constraint imperative programming (class-based objects with inheritance, multi-methods, constraint solving, and dynamically bound constraints) led us to design a special-purpose virtual machine to implement Kaleidoscope. An alternative would have been to use one of the many existing virtual machines for imperative or constraint-based languages.

It would be possible, though extremely awkward, to implement CIP languages with a virtual machine from a conventional object-oriented language with a value-based data store, e.g., the Smalltalk-80 VM [Goldberg & Robson 83]. However, to do so, the Kaleidoscope compiler would have to implement the entire constraint-solving 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 could 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, though awkward, to implement CIP languages using a virtual machine for a pure constraint language or constraint logic language. For example, CLP($\Re$) is a constraint logic programming language whose implementation has a constraint solving engine for constraints over the real numbers [Jaffar et al. 92b]. The CLAM [Jaffar et al. 92a] is the abstract machine used in the CLP($\Re$) interpreter, which is based on the WAM, often used in Prolog implementations [Warren 83], [Ait Kaci 90]. To implement a CIP language using the CLAM, one would have to translate the CIP language semantics into one of the object-oriented logic programming schemes. (We in fact did this in a Kaleidoscope interpreter written in CLP($\Re$). Implementing this interpreter was very useful in exploring language design issues. However, its performance was much worse than that of our current implementation.)

Another implementation technique is to compile into some host language and place embedded calls to the constraint solver library when necessary. CIP languages such as Kaleidoscope use constraints for all computations, so that these embedded calls would be ubiquitous, resulting in a huge, bloated executable file. This approach is ill-suited for CIP languages, but would very likely be the most practical choice for imperative languages with constraint-based libraries.

Still another approach would be to implement a constraint imperative class library. Certain coding rules would be enforced on programmers, such as requiring them to notify the constraint solver after each destructive assignment. Not only would this be inconvenient for programmers, but the compiler used under this approach would be the host language compiler rather than a specialized CIP compiler. Thus there would be no possibility for compile-time analysis to pre-solve or optimize constraints and thus eliminate costly run-time constraint solving. Such analysis is essential for the implementation of high-level languages, and thus we believe that a class library would be the wrong implementation technique.[3]

The cc family of languages [Saraswat 93] generalize the CLP scheme to include such features as concurrency, atomic tell, and blocking ask; if we used such a language instead of Kaleidoscope for constraint programming we could represent objects as perpetual processes that consume an (unbounded) stream of messages. However, the logic programming translations are too unconventional for the present project of modeling objects as mutable entities with state and identity—here we have consciously chosen to be more evolutionary, and thus our goal is to *extend* the imperative framework with constraints rather than asking our clients (Kaleidoscope programmers) to learn a new paradigm.

In previous work we presented language designs for Kaleidoscope'90, '91, and '93 [Freeman-Benson 91], [Freeman-Benson & Borning 92], and [Lopez et al. 93]. Further information on the incremental local propagation algorithms used in this implementation can be found in [Freeman-Benson et al. 90] and [Sannella 93]. Other constraint-based languages include Bertrand [Leler 87] and Siri [Horn 92b], [Horn 92a]. 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 such as long-lived constraints between arbitrary objects.

3. Note that by compile-time analysis we are only specifying that the analysis be done by the compiler, but not when the compiler is run. Thus we are not excluding the option of dynamic compilation as is done in Smalltalk and Self.

6. Conclusions and Future Work

Our virtual machine for CIP languages—the K-machine—is powerful enough to support the unique features of such languages efficiently. The K-machine replaces the value-based data store of a conventional imperative machine with a constraint-based data store. It incrementally maintains this data store, re-satisfying constraints only when necessary. The constraint-based data store utilizes constraint constructors and constraint templates to enforce constraints over objects whose part-whole structure changes dynamically. Constraint constructors are multi-method dispatched, and preserve object encapsulation by not accessing any variables outside the object being constrained.

The Kaleidoscope'93 implementation described in this paper is in use, and we have written a small number of programs to exercise various CIP language features. We plan to continue work on the implementation, to write larger programs in the language, and to feed the results back into 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 run-time constraint satisfaction when possible. (Eliminating runtime constraint satisfaction not only eliminates the cost of solving the constraints, but it also eliminates the need for, and thus cost of, maintaining backpointers and other constraint specific data structures.) Finally, we are designing a constraint-based type system suited for constraint imperative programming languages.

Acknowledgments

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. Thanks also to Ralph Johnson and the other OOPSLA reviewers, and to Craig Chambers, Jeff Dean, Susan Eggers, David Keppel, and Michael Sannella, for their valuable comments and suggestions on various drafts of this paper.

This work has been supported in part by the U.S. National Science Foundation under grants IRI-9102938, IRI-9302249, and CCR-9402551, by the Canadian National Science and Engineering Resource Council under grant OGP-0121431, by a Fellowship from Apple Computer and a Graduate Research Assistantship from the University of Washington Graduate School for Gus Lopez, and by Academic Equipment Grants from Sun Microsystems.

Bibliography

[Ait-Kaci 90] Hassan Ait-Kaci. The WAM: A (real) tutorial. Technical Report 5, DEC Paris Research Laboratory, Paris, January 1990.

[Borning et al. 92] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992.

[Chambers 92] Craig Chambers. Object-oriented multi-methods in Cecil. In *Proceedings of the 1992 European Conference on Object-Oriented Programming*, pages 33–56, June 1992.

[Cohen 90] Jacques Cohen. Constraint logic programming languages. *Communications of the ACM*, 33(7):52–68, July 1990.

[Colmerauer 90] Alain Colmerauer. An introduction to Prolog III. *Communications of the ACM*, pages 69–90, July 1990.

[Freeman-Benson & Borning 92] Bjorn Freeman-Benson and Alan Borning. The design and implementation of Kaleidoscope'90, a constraint imperative programming language. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, pages 174–180, April 1992.

[Freeman-Benson 91] Bjorn Freeman-Benson. *Constraint Imperative Programming*. PhD thesis, University of Washington, Department of Computer Science and Engineering, July 1991. Published as Department of Computer Science and Engineering Technical Report 91-07-02.

[Freeman-Benson et al. 90] Bjorn Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, January 1990.

[Goldberg & Robson 83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Horn 92a] Bruce Horn. 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*, pages 218–233, Vancouver, British Columbia, October 1992.

[Horn 92b] Bruce Horn. Properties of user interface systems and the Siri programming language. In Brad Myers, editor, *Languages for Developing User Interfaces*, pages 211–236. Jones and Bartlett, Boston, 1992.

[Jaffar & Lassez 87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the Fourteenth ACM Principles of Programming Languages Conference*, Munich, January 1987.

[Jaffar et al. 92a] Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. An abstract machine for CLP($\Re$). In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 128–139, San Francisco, June 1992.

[Jaffar et al. 92b] Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. The CLP($\Re$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.

[Leler 87] William Leler. *Constraint Programming Languages*. Addison-Wesley, 1987.

[Lopez et al. 93] Gus Lopez, Bjorn Freeman-Benson, and Alan Borning. Kaleidoscope: A constraint imperative programming language. In Brian Mayoh, Enn Tougu, and Jann Penjam, editors, *Constraint Programming*. Springer-Verlag, 1993. NATO Advanced Science Institute Series, Series F: Computer and System Sciences. Also published as UW CSE Technical Report 93-09-04.

[Lopez et al. 94] Gus Lopez, Bjorn Freeman-Benson, and Alan Borning. Constraints and object identity. In *Proceedings of the 1994 European Conference on Object-Oriented Programming*, pages 260-279, Bologna, Italy, July 1994.

[Sannella 93] Michael Sannella. The SkyBlue constraint solver. Technical Report 92-07-02, Department of Computer Science and Engineering, University of Washington, February 1993.

[Saraswat 93] Vijay A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[Steele Jr. 90] Guy L. Steele Jr. *Common Lisp: The Language*. Digital Press, Bedford, Massachusetts, second edition, 1990.

[Van Hentenryck 89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, MA, 1989.

[Van Hentenryck et al. 92] Pascal Van Hentenryck, Helmut Simonis, and Mehmet Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58(1–3):113–159, December 1992.

[Warren 83] David H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, California, October 1983.

[Wilson & Borning 93] Molly Wilson and Alan Borning. Hierarchical Constraint Logic Programming. *Journal of Logic Programming*, 16(3 & 4):277–318, July, August 1993.