Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming

Bjorn N. Freeman-Benson
University of Washington -and- Université de Nantes
current address:
University of Washington, Dept. of Comp. Sci and Eng. FR-35
Seattle, WA, 98195, USA bnfb@cs.washington.edu

Abstract

Kaleidoscope is an object-oriented language being designed to integrate the traditional *imperative* object-oriented paradigm with the less traditional *declarative* constraint paradigm. Imperative state changes provide sequencing while declarative constraints provide object relations. A *variables as streams* semantics enables the declarative-imperative integration. A running example is used to illustrate the language concepts—a reimplementation of the MacDraw II dashed-lines dialog box. The example is in three parts: the input channel, using imperative code to sequence through modes; the output channel, using constraints to update the display; and the internal relations, using constraints to maintain the data objects' consistency requirements. The last sections of the paper discuss *views* as a natural result of combining objects with constraints, as well as related and future work.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1990 ACM 089791-411-2/90/0010-0077...$1.50

1 Introduction

Kaleidoscope is an imperative object-oriented language in the style of C++, Emerald, Eiffel, Smalltalk, and others, but this is not why Kaleidoscope is interesting. Kaleidoscope is also a constraint language in the style of Sketchpad, ThingLab, Magaritte, Bertrand, and others, but, again, this is not why Kaleidoscope is interesting—numerous papers have explored these issues, some even in this conference [Borning et al. 87, Maloney et al. 89]. Kaleidoscope is interesting because it is an *integration* of these two paradigms—the imperative object-oriented paradigm, which uses destructive assignment to change state; and the declarative constraint paradigm, which does not.

The justification for Kaleidoscope, as for any new programming language, is that better ways can be found to express certain algorithms and concepts. Kaleidoscope was born from the realization that programmers need to specify two types of relations:

1. Long-lived relations between objects to define the information, consistency, and internal structure of an application—an `Engine` is part-of an `Automobile`; this `String` is a printed representation of that `Integer`.

2. Sequencing relations between program states, and between objects in those states—when the mouse button is pressed, bring this `Window` to the front; the position of an `Automobile` is computed from its previous position plus its current velocity.

A traditional imperative object-oriented language only provides sequencing relations, forcing the programmer to shoulder the burden of ensuring that all internal

and external consistency relations are maintained after each set of assignments. Some languages, such as Emerald [Black et al. 86] and Eiffel [Meyer 88], provide assertions or class invariants to check such consistency relations, but do not synthesize code to maintain them.

On the other hand, most constraint languages have no notion of state or sequencing. Those that do, such as Animus [Duisberg 86] and ThingLab II [Maloney et al. 89], provide a read-only `time` variable and use a hidden system process to store historical values for future access. This informal imperative semantics allows these systems to be used for simple animations and simulations but breaks down when dealing with complex constraints and dynamically changing object-oriented sytems.

Kaleidoscope provides both sequencing and constraints by keeping the states of a variable as a history of object values, similar to the variables of Lucid [Wadge & Ashcroft 85]. Long-lived or *durable* relations are constraints on all values in the history, such as the relation between Celsius and Fahrenheit:

$$\forall t, C_t * 1.8 = F_t - 32$$

Transitory relations, such as assignment, are constraints on the value at a particular time (the room temperature at 10:00am is changed to 28):

$$C \leftarrow 28 \qquad \equiv \qquad C_{10:00\text{am}} = 28$$

Or on some range of values (it is cold all winter):

$$\forall t \in \text{Dec–Feb}, F_t \leq 10$$

Kaleidoscope also separates asserting constraints from advancing the streams. Statements are separated by ";" and the streams are advanced by "#". All constraints between hash-marks hold simultaneously, even when separated by complex statements and procedure calls. And, as pointed out by other researchers, this detailed level of control allows program idioms such as swapping two variables without a temporary:

$$\begin{matrix} \texttt{x <- y;} \\ \texttt{y <- x; \#} \end{matrix} \quad \equiv \quad \begin{matrix} x_{t+1} = y_t \\ y_{t+1} = x_t \end{matrix}$$

As a result of mixing these two paradigms, one might assume that programmers using Kaleidoscope would have to mentally shift gears from their previous expertise in imperative object-oriented programming. Fortunately, this assumption is false, as the Kaleidoscope semantics are a superset of both imperative semantics and constraint semantics. A program can be written completely imperatively, using side-effects to change object state, assignment statements, etc.; or completely declaratively, using only constraints; or using any mixture of the two. Programs from a traditional imperative object-oriented language could be syntactically translated into Kaleidoscope; thus the programming style of Smalltalk or C++ could be used in a Kaleidoscope program. However, strictly imperative Kaleidoscope programs would only be using half the power available. A more useful Kaleidoscope style is to write programs with a combination of explicit sequencing and declarative relations—each paradigm expressing those features of the program to which it is most suited.

Sections 2 and 3 start this paper with a review of the constraint theory used, and the semantics necessary for its integration with imperative objects. Section 4 illustrates some language concepts through a running example—a reimplementation of the MacDraw II dashed lines dialog box. Section 5 discusses *views* as a natural result of combining objects and constraints, and section 6 finishes with a review of related work.

Due to space restrictions, this document deliberately simplifies a number of issues. For a complete description of Constraint Imperative Programming paradigm and the Kaleidoscope language, the reader is referred to [Freeman-Benson 90a] and [Freeman-Benson 90b] respectively or to the author's forthcoming Ph.D. dissertation.

As of the publication deadline (July 1990), the language and its virtual machine have been designed, some example programs have been written, and the implementation of the compiler has begun.

2 Constraints and Constraint Hierarchies

Constraints are system maintained assertions about the program. In general, there are many interrelated constraints in a Kaleidoscope program; it is up to the compiler and run-time system to sort out how they interact and to keep them all satisfied.

Kaleidoscope uses a *constraint hierarchy* [Borning et al. 87, Borning et al. 89] to provide a convenient means for stating relative desires. In a constraint hierarchy each constraint is given a strength: `required`, `strong`, ..., `very_weak` with the stronger constraints completely dominating the weaker ones. Other papers have described the theory of constraint hierarchies, and given algorithms for solving them. Kaleidoscope

uses a constraint hierarchy and the *locally-predicate-better* comparator function. This function defines a potential solution, σ, to be *better* than another, θ, if σ completely satisfies at least one constraint that θ does not.

References [Leler 88] and [Freeman-Benson et al. 90] provide more comprehensive overviews of previous constraint languages and systems.

3 Semantics and Time

The key to integrating the declarative constraint paradigm and the imperative object-oriented one is the definition of a semantics that combines the two. As a starting point, one can loosely characterize the semantics of the imperative and declarative paradigms (this paper is not concerned with formal denotational semantics—a rough characterization will suffice):

Imperative
Each instance variable holds a single value (e.g., a pointer to an object). Also, each instance variable potentially holds a different value after each instruction is executed. However, the value of a variable cannot change unless the instruction explicitly writes to that variable.

Declarative
Each variable holds only one value, i.e., the result of evaluating the program. For functional programs, the least fixed-point; for constraint hierarchies, the best valuation. Time does not advance, and the value of the variable cannot change[†].

Thus, inspired by Lucid's use of infinite streams, Kaleidoscope's Constraint Imperative Programming (CIP) semantics merges the two:

Constraint Imperative Programming
Each instance variable holds a stream, or history, of values. Each value represents the value of the variable at a different instant, with subsequent values representing subsequent instants. Time is virtual and represented by the positive integers. The value of a variable at time t is the result (the best valuation) of the constraints that exist at time t. To prevent various paradoxes, the past is read-only.

[†]Logic variables in logic programs can change by becoming more refined during execution, but they cannot arbitrarily change value.

With the following four definitions, the CIP semantics are a superset of both imperative object-oriented and declarative constraint semantics:

1. The value at time t is, by default, the same as the value at time $t-1$. Thus a variable's value does not change unless the variable is assigned to or otherwise constrained. This is implemented in Kaleidoscope by a `very_weak stay` constraint between the values of the stream:

$$\forall t, \texttt{very_weak}\ V_t = V_{t-1}$$

2. An assignment constraint is a constraint on the next value of the stream and thus can only affect the next instant. It can affect the distant future only when the new value is propagated by the `very_weak stay` constraint mentioned above. It cannot affect the present or the past. For example, if the current time is 8 then:

$$\texttt{x <- y;} \quad \equiv \quad x_9 = y_8$$

3. All other constraint expressions denote a (potentially) infinite set of constraints on individual values in the streams. Thus, in Kaleidoscope, the constraint expression:

$$\texttt{pig} = \texttt{cow}$$

defines a value constraint for each instant:

$$\forall t, \texttt{pig}_t = \texttt{cow}_t$$

or, equivalently, starting at time $t = 1$:

$$\texttt{pig}_1 = \texttt{cow}_1$$
$$\texttt{pig}_2 = \texttt{cow}_2$$
$$\vdots$$

In other words a constraint expression affects the values of the variables it constrains "from now on". The `assert...during...` construct demonstrated in section 4.5 corresponds to "from now until then," or:

$$\forall t \in n \ldots m, \texttt{pig}_t = \texttt{cow}_t$$

And, as discussed earlier,

4. Time is explicitly advanced using the "#" operator. Here Kaleidoscope differs from traditional object-oriented languages which advance time as each byte-code or machine language instruction is executed.

Thus, a program without constraints (only assignments), has similar semantics to an imperative program (values stay the same, assignments change values). A program without assignments and hash-marks

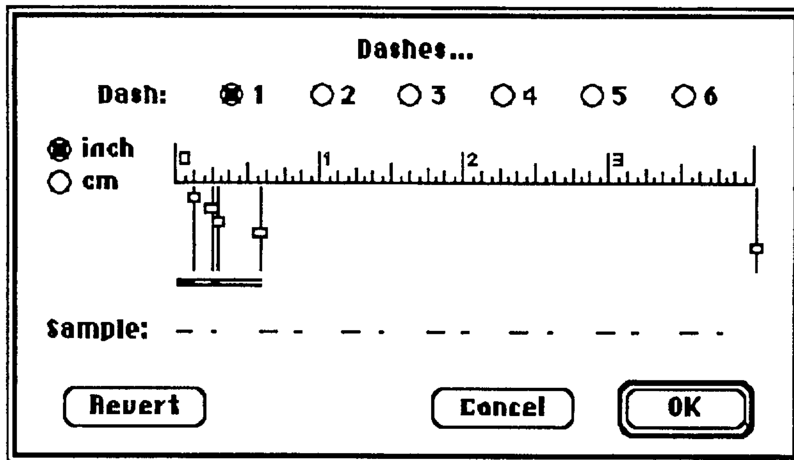

Figure 1: MacDraw II Dashed Line Dialog Box

has the same semantics as a constraint program (variables have a single value; the same single value at each instant).

Naturally, an efficient Kaleidoscope implementation will recognize when a variable is being used strictly imperatively and represent it as a more efficient destructive assignment variable, perhaps even storing it in a register.

4 __ Language Overview __

The running example used to demonstrate the language concepts is a reimplementation of the MacDraw II dashed-lines dialog box seen in figure 1 (thanks to Joel Spiegel for suggesting this example). This dialog box is used to specify the pattern of black and white dashes that make up a dashed line. There are three main components to the interaction:

1. The data consistency constraints—constraints such as "all dashes are adjacent," and "each vertical bar is attached to the end of a dash."
2. The output channel—the routines that maintain the graphical representation of the data structure within the dialog box.
3. The input channel—the routines that handle mouse and keyboard events by translating them into data structure modifications and mode transitions.

A `DragBar` is a vertical line with a small mouse sensitive box. Figure 1 has four `DragBars` for the four dashes (two black and two white) and one extra `DragBar` on the far right. A new dash is created by dragging the far right `DragBar` into the ruler space, and, conversely, the last dash can be deleted by dragging its `DragBar` to the far right (see figures 6 and 7).

The original dialog box source code is about 360 lines of Object Pascal and took two to three weeks to write with about half the time spent dealing with corner cases and constraints [Spiegel 89].

4.1 __ Constraints and Classes __

Kaleidoscope has a standard class-based, garbage-collected approach to objects. Objects can have public and private instance variables and operations (methods). An `initially` clause can be defined to initialize each new object. For example, the `Dash` class has four instance variables, three constraints, and an initial color:

```
class Dash subclass of Object
   public var left, length, color;
   public virtvar right;
   initially
      always: right - left = length;
      always: length >= 1;
      always: length <= 128; % 128 = 1¾"
      weak color <- Black;
   end initially;
end Dash;
```

The `right` variable is defined as a *virtual variable*: a name that acts like a variable but uses no storage. Instead, a virtual variable is defined in terms of other variables and is recomputed upon each access. Because constraints are used, virtual variables can be assigned

to, as well as be read from. A virtual variable must be defined by at least one constraint.

Each constraint in Kaleidoscope has a strength—those without an explit strength are, by default, **required**. Thus far, the example program's constraints have all been **required**. As described in section 3, an assignment is a constraint, thus assignments can also have a strength.

Kaleidoscope uses classes for implementation inheritance. For example, in the following code fragment, **DashedLine** inherits its representation and operations from **Array** and then adds four constraints. Note that the statement labels "*1...*" are not part of the Kaleidoscope syntax:

```
class DashedLine subclass of Array
   initially
1...    always: self[first].left = 0;
2...    always: self[first+1..last].left =
            self[first..last-1].right;
3...    always: self[first].color = Black;
4...    always: self[first+1..last].color =
            self[first..last-1].color.inverse;
   end initially;
end DashedLine;
```

These four constraints are *iterative* constraints: constraints on objects whose size changes dynamically (in this case **self**, a subclass of **Array**). Constraint 1 places the first dash at the left edge of the line, 2 places each dash adjacent to its neighbors, 3 makes the first dash **Black**, and 4 causes the colors to alternate:

4.2 — Types of Constraints —

The time complexity of constraint satisfaction depends on both the domain and kind of the constraints. For example: linear equations over the real numbers can be solved in polynomial time, single polynomials to degree four have a closed form solution, discrete CSP problems are NP-complete, and the complexity for integer polynomials of degree greater than two is still unknown. As a result, much of the effort in constraint language research has gone into finding algorithms, heuristics, and language subsets that can be solved efficiently. In Kaleidoscope, these limitations manifest themselves in how the constraint solver operates.

For example, because Kaleidoscope is essentially an imperative language, it does not support backtracking. Thus the solver uses a committed-choice approach: once a value has been determined, further constraints in the same time interval cannot change its value. Thus, if an additional constraint producing a different value is encountered before the next time advance, the program will halt with a run-time error. For example, the following code fragment is illegal:

```
if X > 5 then
   once: X = 3;
end if;
```

The reader is referred to [Freeman-Benson 90a] for complete details.

Thus there are three types of constraints in Kaleidoscope:

Equality
: Value equality can be both *soft* (user-defined) and *hard* (system-defined).

Primitive
: The core of the constraint solver only solves primitive constraints over real numbers, integers, and booleans using a few pre-defined operators (+, −, =, <, ...). Primitive constraints may be cyclic, redundant, and even uni-directional. Note that the solver cannot perform miracles—it cannot solver Fermat's last theorem—however it will have a number of algorithms to solve various kinds of constraints over various domains.

Complex
: Complex constraints are constraints over complex objects built by *constraint constructors* (special side-effect-free procedures) from lower-level constraints. For example, **Point** addition can be constructed as:

```
class Point
   constructor +( q, r )
      always: self.x + q.x = r.x;
      always: self.y + q.y = r.y;
   end +;
end Point;
```

4.3 — Complex Constraints —

The dashed lines dialog box allows six different dashed lines to be specified, each with a different arrangement of dashes. In this reimplementation, a **DashPatternInteraction** object is created when one of the six radio buttons is selected. The **DashPatternInteraction** handles

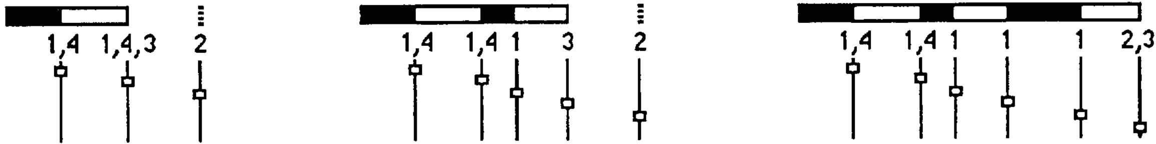

Figure 2: Minimum, Median, and Maximum Configurations

the user interaction, returning control only when the mouse is clicked outside its sphere of influence.

```
class DashPatternInteraction
   var draggers, line;
   initially
      line <- DashedLine.new();
      draggers <- Array.new();
      ...
```

The number of DragBars is one more than the number of Dashes in the DashedLine because of the extra DragBar waiting on the right. However, there is a maximum of six dashes in a pattern, thus when six dashes are present, the extra DragBar should not be. This constraint could be written:

```
always: draggers.size =
       min( line.size+1, 6 );
```

But writing the constraint this way does not specify how to modify the arrays, i.e., what object to place in new array entries. Instead, this constraint is written using a constraint constructor in the DashedLine class. Note that this constraint is asymmetric: if line changes size, draggers will adapt; but not vice versa.

```
always: line.min_six( draggers );

class DashedLine
   constructor min_six( d )
      var diff;
      once: diff = d.size
          - min(self.size+1,6);
%
% If I am the bigger array then
% add to the draggers array.
      if diff < 0 then
          d.add_n_last(
             DragBar.new(), -diff );
% If I am smaller, remove from draggers.
      elseif diff > 0 then
          d.remove_n_last( diff );
      end if;
   end min_six;
end DashedLine;
```

As a general rule, the DragBars are attached to endpoint of their corresponding Dash (constraint 1 below), and the last DragBar rests at the right edge of the dialog box (constraint 2). However, there are two special cases: first, the DragBar of the last Dash is only weakly attached so that the DragBar may be dragged to the far right (constraint 3); and second, there must be at least two dashes in each pattern (a dashed line with one dash is a solid line!). Thus, if the last dash is the first or second dash, it cannot be dragged to the far right (constraint 4). This is very confusing in words, but is easy to express in Kaleidoscope:

```
1...always: required
     draggers[..last-1].offset
     = line.dashes[..].right;
2...always: very_weak
     draggers[last].offset = RHS;
3...always: weak
     draggers[line.dashes.size].offset
     = line.dashes[last].right;
4...always: required
     draggers[1..2].offset
     = line.dashes[1..2].right;
```

Figure 2 shows which constraints connect to which DragBars in various situations. As explained in section 2, required > weak > very_weak, so that if the constraints conflict, the stronger ones (1 and 4) will override the weaker ones (2 and 3). The constraints can be specified in any order.

4.4 — Output Channel —

The classes and constraints defined in the previous sections are the data objects and internal consistency

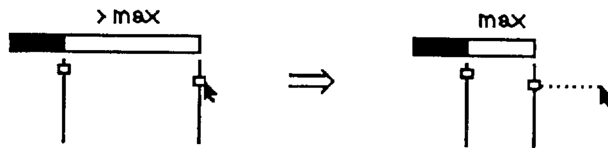

Figure 3: Attempting to overstretch a Dash

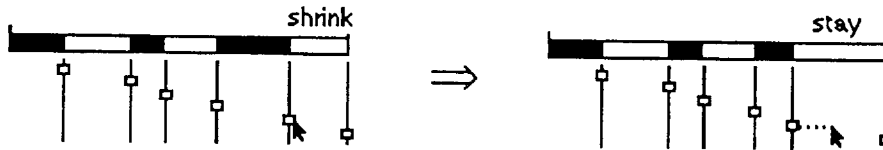

Figure 4: Attempting to compress a Dash

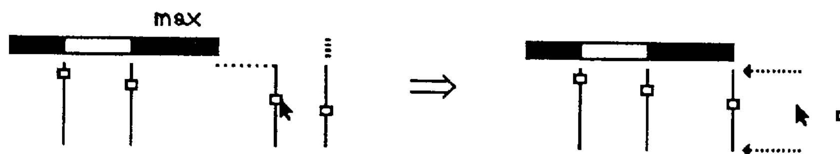

Figure 5: The DragBar snaps back

relations for the dialog box. To define the output channel (alternatively, the output relation), a Kaleidoscope program uses constraints as one-way *filters* from source objects to bitmaps [Ege et al. 87]. In other words, the constraints are being used as a uni-directional functional language similar to Fabrik [Ingalls et al. 88]. The filter constraints are uni-directional because: (1) there is no good algorithm to determine the positions of the DragBars from arbitrary bitmaps, and (2) the committed-choice approach of the solver cannot execute the filter constraints in reverse. The actual code is not shown.

4.5 — Input Channel —

The last part of the example is the input channel. The interesting part of the input channel is the procedure that handles a mouse-down event in the drag box of an DragBar. The procedure has two parts: a set of constraints that temporarily change the system's behavior, and a block of imperative code to sequence through the three modes: dragging, off-to-the-right, and finished. The constraints are:

1. The DragBar position is constrained to follow the mouse position, but only with a medium strength. Thus, in figure 3, when the mouse attempts to drag the DragBar into a position that would violate other stronger constraints, the DragBar does not follow.
2. The length of all Dashes, except the selected one, are required to remain fixed, i.e., to stay the same as their previous value. If they were not so constrained, or constrained at a weaker level, then the medium mouse constraint could change their length by pushing them up against the edges of the dialog box. See figure 4.
3. The length of the selected Dash is weakly constrained so that the mouse can stretch it—recall that weak is weaker than the mouse constraint's medium.

These constraints are placed in an assert...during... construct to assert that the temporary constraints should only be valid during the duration of the enclosed block.

```
    assert
1...    medium draggers[sel].offset =
            mouse.position.x;
2...    required line[..\sel].length stay;
3...    weak line[sel].length stay;
    during
        ...
```

The duration of the dragging interaction is enclosed in the during block, and is defined by a loop that terminates when the mouse button is released. During the dragging, any of three interesting events can happen:

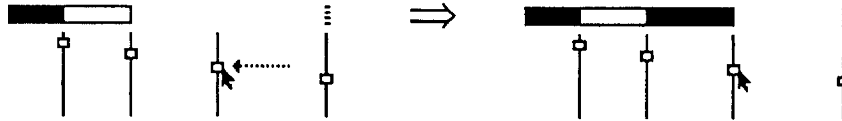

Figure 6: Creating a dash

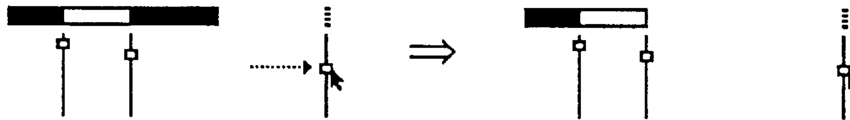

Figure 7: Deleting a Dash

4. The mouse button can be released, terminating the dragging, and removing the temporary constraints. As shown in figure 5, if the last `DragBar` has been dragged into the blank space between the last `Dash` and the right edge of the box, the constraints will snap it back to the end of its corresponding `Dash`.

5. The `DragBar` can be dragged from the far right, creating a `Dash` (figure 6).

6. The `DragBar` can be dragged to the far right, deleting a `Dash` (figure 7).

The code is as follows:

```
        loop
            if mouse.button = false then
4...            exit;
            elseif draggers[sel].offset <> RHS
                and sel = line.size+1 then
5...            line.add_last( Dash.new );
            elseif draggers[sel].offset = RHS
                and sel <> line.size+1 then
6...            line.remove_last();
            end if; #
        end loop;
    end assert;
```

The remainder of the dialog box program is not shown.

5 — Views

Kaleidoscope gets its name from an original goal of using constraints to support multiple views in object-oriented programming. In traditional object-oriented languages, each object presents the same aspect of itself to all viewers, or rather to everyone except itself—the `self` view often extends the public view with a private protocol and instance variables. In C++, the `friend` and `protected` keywords allow an object to have three views: private, protected, and public. Numerous research projects including [Goldstein & Bobrow 80, Wadler 87, Haberman et al. 88, Shilling & Sweeney 89, Harrison et al. 89] have provided single objects with multiple interfaces.

For example, an `Employee` might also be viewed as a `Friend`, and perhaps as a `BloodDonor`. Some of the instance variables of the three views are shared, whereas others are not:

Employee	Friend	BloodDonor
name	name	name
age	-	age
ssn#	-	-
-	-	blood_type

A more traditional computer science example is viewing a `Tree` data structure as an `Array` (with a pre-order filter), or as a `Set` (by an any-order filter). The definition, use, and implementation of views is an active area of research.

The simplest view of an object is as a supertype: an `Integer` as a `Number`. Following the Emerald terminology, this type of viewing is referred to as *narrowing*[†] the view of an object because it reduces the number of operations that can be invoked on the object. Conversely, *widening* views an object as a subtype. Widening is safe if the object is really just a narrowed view of the subtype, e.g., `(4 as Number) as Integer`.

[†]Not the same definition of narrowing as in [Reddy 85].

However, if the object being widened is *not* the appropriate type, then the new view will contain information not present in the old view, and the problem becomes the view update problem of databases [Date 81]. Traditional languages often catch this error with a run-time trap, for example the Smalltalk `doesNotUnderstand:` message.

In Kaleidoscope, a view is an object (or equivalently, an object is a view). Each view of some information (each view of a particular object) is a *separate* object†. The objects are connected by consistency constraints—often equality constraints between their common instance variables, although any other constraints are permitted. The viewing operation consists of creating the new object (instantiating its class) and creating the consistency constraints. Thus the viewing operation "**as**" is no different than a standard operation. For example:

```
class Employee
    var name, ssn;
    public func as( Friend ) -> ( f )
        f <- Friend.new; #
        always: f.name = self.name;
    end as;
end Employee;
```

Because the object-oriented paradigm places the operations (methods) in each object, writing corresponding views between two classes requires a lot of duplication. For example, the programmer must write one function for **Celsius as Fahrenheit** and another for **Fahrenheit as Celsius**, each containing the constraint $C * 1.8 = F - 32$. To ease this burden, Kaleidoscope provides a view definition syntax that, in effect, automatically generates an **as** function for each class:

```
view Celsius <--> Fahrenheit <--> Kelvin
    always: Celsius * 1.8 = Fahrenheit - 32;
    always: Celsius = Kelvin + 273;
end view;
```

Constraints in the view definition can have different strengths, and assignment constraints are permitted to give default values:

†Conceptually, at least, a view is a separate object. The compiler may choose to implement multiple views as a single object using techniques from [Haberman et al. 88, Shilling & Sweeney 89].

```
    view Point <--> Pixel
        always: Point.x = Pixel.x;
        always: Point.y = Pixel.y;
1a...   weak Pixel.color <- Grey;
    end view;
```

Note that the apparently similar view definition:

```
    view Point <--> Pixel
        always: Point.x = Pixel.x;
        always: Point.y = Pixel.y;
1b...   weak Pixel.color = Grey;
    end view;
```

actually specifies a different behavior. Constraint 1b defines a **weak** equality that overrides the `very_weak` $V_t = V_{t-1}$ value propagation **stay** constraint mentioned in section 3. Thus the `color` would return to `Grey` after each assignment statement:

```
pixel <- point as Pixel; #
    % pixel.color is Grey
pixel.color <- White; #
    % pixel.color is White
#
    % pixel.color is Grey again
```

The multiple activations of [Shilling & Sweeney 89] allow multiple views of the same class: a **Friend** may have three jobs, and thus three **Employee** views. The current definition of Kaleidoscope supports only a single view of each class (e.g., a **Friend** can only have one **Employee** view), although the extension to labelled or query-selected multiple views is being considered for the future.

6 — Related Work

The previous and related work can be grouped into three categories: object-oriented languages, constraint based systems, and combinations thereof.

Kaleidoscope is a dynamically-typed object-oriented language, and with the exception of its viewing mechanism, Kaleidoscope does not present any new ideas in its basic object-oriented features. The debates on metaclasses versus prototypes, inheritance versus delegation, static typing versus dynamic typing, control flow with messages versus control flow with statements, compiled versus interpreted, etc. are all relatively independent of the interesting issues that Kaleidoscope explores—those problems and opportunities

that arise when integrating declarative and imperative object-oriented paradigms. Consequently, the Kaleidoscope design has chosen to use classes without metaclasses, run-time typing, statement-based control flow, and a mixture of compilation and interpretation because these techniques are useful and well-understood, rather than because they are the "best."

Previous and continuing constraint language research in a variety of fields has demonstrated the utility of multi-directional declarative programs. Bibliographies and discussions can be found in [Leler 88] and [Freeman-Benson et al. 90]. In Constraint Logic Programming (CLP) [Jaffar & Lassez 87], its cousin HCLP [Borning et al. 89], and the cc family of languages [Saraswat 89], logic programs are used to construct and query constraint networks. Similarly, Kaleidoscope uses an imperative program to construct and query a network. Logic programming supports backtracking; Kaleidoscope does not.

The Kaleidoscope combination of constraints and object-oriented programming was inspired by the infinite streams of Lucid [Wadge & Ashcroft 85] as well as by the Fabrik visual programming environment [Ingalls et al. 88]. Independently inspired, [Berlandier & Moisan 88] developed a similar constraints-and-sequencing system named Prose, although without complex constraints. Other constraints-plus-object-oriented languages, e.g., [de Wegher et al. 89, Avesani et al. 90], have not addressed the issues of constraints on dynamically changing objects nor on objects at different part-subpart levels. Instead, these languages only permit constraints on terminal variables. Kaleidoscope uses constraint constructors to, in effect, *split* a larger constraint on an object into smaller constraints on its subparts.

Kaleidoscope and Prose are similar in that they both use streams of values to hold state. Kaleidoscope uses a *constraint hierarchy* to provide a clean semantics for default information, and emphasizes efficiency. Prose is a constraint language within the SMECI object-oriented expert system, and relies on a backtracking solver to handle conflicts and higher-order constraints. The constraint language and expert system are separate yet interconnected: the supervising expert system can add and remove constraints as the rules are fired; and the constraints can fail, causing the expert system to backtrack. Kaleidoscope, on the other hand, uses constraints to enforce rather than to verify relations. Thus, instead of being controlled by an external system that backtracks, the Kaleidoscope system maintains the constraints during the natural execution of the program.

The use of a constraint hierarchy rather than a flat constraint system is essential in the semantics of Kaleidoscope because it allows the imperative "variables stay the same unless changed" and "assignment changes the value" relations to be defined in terms of constraints (cf. section 3). Systems without a hierarchy cannot cleanly integrate these two paradigms.

7 Conclusion

This paper has outlined some of Kaleidoscope's features for integrating declarative constraints and imperative object-oriented programming. These features have been selected to provide a useful yet efficient language. As with any new language, there are a number of rough spots. The obvious solution to these problems is to allow maximal generality. However, the language designer must keep in mind that a high-level language should not penalize the programmer. Powerful features will always consume resources, but they should only do so when actually used. In other words, a Kaleidoscope program without constraints should be as fast as the corresponding C++ program, as should a Kaleidoscope program with constraints that can be statically compiled. Only in complex situations should the Kaleidoscope compiler resort to using run-time constraint solver calls. Useful implementation techniques have been gathered from many previous systems, including [Chambers et al. 89, Graver 89, Chambers & Ungar 90, Freeman-Benson 89] and are being used to build a prototype Kaleidoscope system.

Mixing constraints, objects, and imperative programming has proved useful in ThingLab II, and in the sample Kaleidoscope programs written so far. The dialog box example in section 4 demonstrates the utility of using constraints to define relationships between objects in three parts of a user interface: the input channel, the output channel, and the internal data invariants. Other programs have demonstrated that adding constraints to an object-oriented program is useful programming technique. And last, but not least, adding explicit objects, state, and time to a constraint language allows a compact description of many sequencing algorithms. Kaleidoscope is a language explicitly designed for all these techniques.

Acknowledgements

Thanks to Alan Borning and Michael Sannella for help with the language, to Jean Bézivin, Craig Anderson,

Kirsten Freeman-Benson, and the anonymous referees for comments on this paper, and to Joel Spiegel for suggesting the example. This research has been supported by a Chateaubriand Fellowship from the French Government, an ELF Aquitaine grant, a fellowship from the National Science Foundation, the National Science Foundation under Grant No. IRI-8803294, and by the Washington Technology Center.

References

[Avesani et al. 90] P. Avesani, A. Perini, and F. Ricci. COOL: An Object System with Constraints. In *TOOLS 2*, June 1990.

[Berlandier & Moisan 88] Pierre Berlandier and Sabine Moisan. Reflexive Constraints for Dynamic Knowledge Bases. In *Proceedings of the 1st International Computer Science Conference '88: Artificial Intelligence: Theory and Applications*, Hong-Kong, December 1988.

[Black et al. 86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object Structure in the Emerald System. In *OOPSLA'86 Proceedings*, pages 78–86, Portland, Oregon, September 1986. ACM.

[Borning et al. 87] Alan Borning, Robert Duisberg, Bjorn Freeman-Benson, Axel Kramer, and Michael Woolf. Constraint Hierarchies. In *OOPSLA'87 Proceedings*, pages 48–60. ACM SIGPLAN, October 1987.

[Borning et al. 89] Alan Borning, Michael Maher, Amy Martindale, and Molly Wilson. Constraint Hierarchies and Logic Programming. In *Proceedings of the Sixth International Logic Programming Conference*, pages 149–164, Lisbon, Portugal, June 1989.

[Chambers & Ungar 90] Craig Chambers and David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In *Proceedings of the SIGPLAN Programming Language Design and Implementation.* ACM SIGPLAN, June 1990.

[Chambers et al. 89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes. In Norman Meyrowitz, editor, *OOPSLA'89 Proceedings*, pages 49–70, New Orleans, Louisiana, October 1989. ACM SIGPLAN.

[Date 81] C. J. Date. *An Introduction to Database Systems.* Addison-Wesley Publishing Company, New York, NY, 1981. Third Edition.

[de Wegher et al. 89] Isabelle de Wegher, Jean-Marc Trinon, and Eric Meirlaen. Reinforcing Consistency in an Object Oriented System by the use of a Constraint Language. In *TOOLS'89 Proceedings*, pages 253–260, Paris, France, November 1989.

[Duisberg 86] Robert Duisberg. *Constraint-Based Animation: The Implementation of Temporal Constraints in the Animus System.* PhD thesis, University of Washington, 1986.

[Ege et al. 87] Raimund Ege, David Maier, and Alan Borning. The Filter Browser—Defining Interfaces Graphically. In *ECOOP'87 Proceedings*, pages 155–165, Paris, June 1987. Association Française pour la Cybernétique Économique et Technique.

[Freeman-Benson 89] Bjorn Freeman-Benson. A Module Compiler for ThingLab II. In *OOPSLA'89 Proceedings*, New Orleans, October 1989. ACM.

[Freeman-Benson 90a] Bjorn Freeman-Benson. The Evolution of Constraint Imperative Programming. Technical Report 90-05, Laboratoire d'Informatique, Université de Nantes, Nantes, France, July 1990.

[Freeman-Benson 90b] Bjorn Freeman-Benson. The Kaleidoscope Programming Language: A Second Report. Technical Report 90-06, Laboratoire d'Informatique, Université de Nantes, Nantes, France, July 1990.

[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.

[Goldstein & Bobrow 80] Ira Goldstein and Daniel Bobrow. A Layered Approach to Software Design. Technical Report CSL-80-5, Xerox PARC, December 1980.

[Graver 89] Justin Owen Graver. *Type-Checking and Type-Inference for Object-Oriented Programming Languages.* PhD thesis, University of Illinois, Urbana-Champaign, August 1989.

[Haberman et al. 88] A. N. Haberman, Charles Krueger, Benjamin Pierce, Barbara Staudt, and John Wenn. Programming with Views. Technical Report CMU-CS-87-177, CMU, January 1988.

[Harrison et al. 89] William H. Harrison, John J. Shilling, and Peter F. Sweeney. Good News, Bad News: Experience Building a Software Development Environment Using the Object-Oriented Paradigm. In Norman Meyrowitz, editor, *OOPSLA'89 Proceedings*, pages 85–94, New Orleans, Louisiana, October 1989. ACM SIGPLAN.

[Ingalls et al. 88] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle. Fabrik: A Visual Programming Environment. In *OOPSLA'88 Proceedings*, pages 176–190. ACM SIGPLAN, September 1988.

[Jaffar & Lassez 87] Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *Proceedings of the 14th ACM Principles of Programming Languages Conference*, Munich, January 1987. ACM.

[Leler 88] William Leler. *Constraint Programming Languages: Their Specification and Generation.* Addison-Wesley Publishing Company, 1988.

[Mackworth 77] Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[Maloney et al. 89] John Maloney, Alan Borning, and Bjorn Freeman-Benson. Constraint Technology for User-Interface Construction in ThingLab II. In *OOPSLA'89 Proceedings*, New Orleans, October 1989. ACM.

[Meyer 88] Bertrand Meyer. *Object-oriented Software Construction.* International Series in Computer Science. Prentice Hall, 1988.

[Reddy 85] Uday S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proceedings of the 1985 Symposium on Logic Programming*, pages 138–151, Boston, Massachusettes, July 1985. IEEE.

[Saraswat 89] Vijay Anand Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, Computer Science Department, January 1989.

[Shilling & Sweeney 89] John J. Shilling and Peter F. Sweeney. Three Steps to Views: Extending the Object-Oriented Paradigm. In Norman Meyrowitz, editor, *OOPSLA'89 Proceedings*, pages 353–362, New Orleans, Louisiana, October 1989. ACM SIGPLAN.

[Spiegel 89] Joel Spiegel, July 1989. Personal Communication.

[Wadge & Ashcroft 85] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language.* Academic Press, London, 1985.

[Wadler 87] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM Principles of Programming Languages Conference*, Munich, January 1987. Revised version.