The Design and Implementation of Kaleidoscope'90, A Constraint Imperative Programming Language

Bjorn N. Freeman-Benson
University of Victoria
Department of Computer Science
Box 3055
Victoria, B.C. V8W 3P6
CANADA
bnfb@csr.uvic.ca

Alan Borning
Department of Computer Science
and Engineering, FR-35
University of Washington
Seattle, Washington 98195
USA
borning@cs.washington.edu

Abstract

Two major paradigms in computer programming languages are imperative *and* declarative *programming. We describe a scheme for languages that integrate specific features from these two paradigms into a new framework: Constraint Imperative Programming. Along with the framework, we discuss the design and implementation of a particular instance of this framework, Kaleidoscope'90. From the imperative paradigm, constraint imperative programming adopts explicit control flow, state, and assignment. From the declarative paradigm, it adopts explicit, system-maintained constraints (relations that should hold). There is a strong practical motivation for making this integration: in a typical application, some portions are most clearly described using imperative constructs, while other portions are most clearly described using constraints. By using a constraint imperative language, the most suitable paradigm can be used as appropriate.*

1 Introduction

We propose to combine two simple techniques—declarative *constraint* programming and object-oriented *imperative* programming—to construct a programming language that allows programming of a new kind. A constraint is a system-maintained relation, for example, that $a + b = c$, that the width of a screen rectangle be twice its height, or that the colors in two rectangles on the screen be the same. Imperative programming includes state, destructive assignment, and sequencing. The variety of imperative programming that we use in our design and implementation is object-oriented, which adds both additional power and complications.

We are motivated to integrate these techniques in one language because, in a typical application, there are some portions that are most clearly and conveniently described using constraints, and other portions that are most clearly described using imperative constructs. This integration is difficult because of the fundamental differences between the two paradigms—the very same dichotomy that makes the result interesting and useful. These basic differences are in the storage model, the computation model, the flow of control, and the data types.

In this paper we describe these differences in more detail, and then discuss a framework, Constraint Imperative Programming (CIP), that integrates the two paradigms. We also present a new programming language, Kaleidoscope'90, that is an instance of this framework. Kaleidoscope'90 has been implemented, and various sample programs have been executed. This paper is based primarily on Freeman-Benson's recently completed Ph.D. dissertation [11]. Reference [7] is a preliminary conference paper on Kaleidoscope, while [8] discusses in detail issues that arise when integrating objects and constraints.

2 A Motivating Example

Consider the scroll bar shown on the left. The scroll bar has "up" and "down" buttons and a "thumb." The size of the thumb is proportional to the number of lines of the document that are visible in the scrolled window. Pressing the up button scrolls the text up one line; similarly for the down button. If the mouse is clicked in the gray region above or below the thumb, the scroll bar moves by one page. Finally, the thumb can be dragged

directly with the mouse to a new position.

Properties of this scroll bar that are conveniently described using constraints include:

- the position of the top of the thumb is proportional to the number of lines of the displayed document that are scrolled off the top of the window. This constraint may be coded as:

```
always: thumb.top = bar.top -
        bar.height *
        (document.first_visible /
          document.total_lines);
```

- the height of the thumb itself is proportional to the percentage of the document that is visible in the window:

```
always: thumb.height = bar.height *
        (document.visible_lines /
          document.total_lines);
```

- the area within the thumb is colored white on the display (other constraints control the scrollbar's visibility as its window is overlapped or clipped):

```
always: thumb.interior.color = White;
```

We would like these constraints to be multidirectional and automatically maintained. For example, if the user drags the thumb of the scroll bar, the position constraint mentioned above would be used in one direction to scroll the document (`document.first_visible := ... thumb.top ...`), whereas if some other activity scrolls the document, the constraint would be used in the other direction to update the thumb position (`thumb.top := ... document.first_visible ...`). Further, the programmer should be able to simply state the third constraint, and leave it to the underlying compiler and runtime system to ensure that the graphical appearance of the thumb is maintained correctly.

Other properties of the scroll bar are more conveniently described using imperative constructs, for example, the object's behavior when the user clicks on the up button:

```
if inside(up_button, mouse.location) then
  document.first_visible :=
    document.first_visible -
    document.one_page;
end if;
```

Finally, some properties are most succinctly described using a combination of constraint and imperative constructs. For example, if the mouse button is pressed while the mouse is within the thumb rectangle, then as long as the button is held down, a constraint that the thumb position tracks the mouse position should be active; when the button is released this constraint becomes inactive. Naturally, during this interaction, all other constraints are maintained, in particular the display constraints that keep the screen image up-to-date. This behavior is coded as:

```
if ( mouse.button = down ) then
  if inside(thumb, mouse.location) then
    offset := thumb.center - mouse.location;
    while ( mouse.button = down ) assert
```

```
      medium thumb.center = mouse.location
                            + offset;
    end while;
  end if;
  ...
end if;
```

3 Integrating the Paradigms

As noted in the introduction, integrating constraint and imperative programming is difficult because of the fundamental differences between the two paradigms. These differences are in the storage model, the computation model, the flow of control, and the data types.

3.1 The Donor Paradigms: Imperative and Constraint

The imperative paradigm is based on the Von Neumann model of a stored program computer: an instruction processor and a data store.[1] Each location in the value store is mapped to exactly one value. The value store can only be updated by a destructive write. Further, because each location can potentially contain a different value at any given time, we can consider the value store as a function $Time \times Location \rightarrow Value$. The instruction processor fetches and executes four basic kinds of instructions: *read* a value from the value store, destructively *write* a value to the value store, *compute* some function of values, and *branch* (conditionally or unconditionally) to another location in the instruction store.

The constraint paradigm *per se* has no instruction processor or data store. Rather, a constraint program can be represented by a hyper-graph in which the nodes are variables and hyper-edges are constraints. Each variable holds a set of values: the set of all values that are consistent with the constraint graph. However, this set is constant for a given constraint graph, i.e., the only mechanism for changing the set of values is to operate on the graph from some external driver program. The graph can thus be considered as a function $Location \rightarrow SetOfValues$ where *Location* is a variable name.

Constraint imperative programming is thus an integration of the active portions of its two donor paradigms: the imperative paradigm consists of an active instruction processor and a passive value store, whereas typical constraint systems consist of an external interface routine and an active constraint graph/value store. The CIP paradigm combines the two by replacing the imperative paradigm's passive value store with the constraint paradigm's active constraint graph, and the constraint paradigm's interface routine with the imperative paradigm's active instruction processor.

The constraint paradigm can be extended to allow both *required* and non-required, or *preferential*, constraints. The required constraints must hold for all solutions, while the preferential constraints should be satisfied if possible, but no error condition arises if they are not. A constraint hierarchy can contain an arbitrary number of levels of preference, in either a total order or a partial order, as long as the stronger constraints completely dominate the weaker ones. The solutions to a constraint hierarchy are the valuations (mappings from variables to values) that best satisfy the constraints in the hierarchy, respecting their relative strengths. An important use of constraint hierarchies in constraint imperative programming is to express weak equality constraints between successive states of a variable, so that its value doesn't change gratuitously. (This issue is discussed in more detail in the next section.) However, constraint hierarchies have many other uses as well, for example, to express default initial values, or preferences in window layout or in a scheduling application. For example, the following constraint states that the default background color of the display is gray:

```
always: weak Display.color = Gray;
```

Stronger constraints (such as the constraint on the color of the thumb in the scrollbar example) would locally override this weak constraint.

References [2, 11] present a formal description of the theory of constraint hierarchies, while [9, 10] describe two algorithms, DeltaBlue and DeltaStar respectively, for finding solutions to them.

3.2 Storage Model Integration

Integrating the time aspect of the imperative storage model and the constraint graph aspect of the constraint storage model results in a value store defined as a sequence of constraint graphs, one for each time interval. We can consider this store as a function $Time \times Location \rightarrow SetOfValues$. For notational convenience, we refer to a $Time \times Location$ pair as a *pellucid variable*[2] and write it as x_t. Thus the store maps $x_t \mapsto SetOfValues$. If all the constraints in the store have the restricted form $x_t = c$ for some constant c, then the store behaves like an imperative store; if time

[1] Without loss of generality, and to simplify the subsequent discussion, we assume that the data store is divided into an instruction store and a value store.

[2] Pellucid: transparent, translucent. Pellucid values are useful for discussing the semantics of CIP, but cannot be directly accessed by a CIP program. Thus they are "transparent or translucent."

Statement	*Constraint Template*
`always: c * 1.8 = f - 32.0;`	$\forall t, \quad c_t * 1.8 = f_t - 32.0$
`once: c = -40.0;`	$t = 9, \quad c_t = -40.0$
`while ( mouse.button = down ) assert c = mouse.location.x;`	$t \in 9 \ldots 15, \quad c_t = mouse.location.x_t$

Figure 1: Constraint Template Examples

is ignored, the store behaves like a constraint graph. We have also found it useful to visualize the variables in a constraint imperative program as streams of pellucid variables, i.e., $x \mapsto \langle x_1, x_2, x_3, \ldots \rangle$. (This way of visualizing variables as holding streams of values is taken from Lucid [27].)

In a constraint program, adding a constraint to the constraint graph asserts the constraint for the duration of the program (because there is only one such graph). However, in a CIP program, there is a separate graph for each time interval (at least conceptually). Thus, to assert a constraint for the duration of the program, the same constraint must be added to each constraint graph. This can be accomplished by keeping a table of constraint abstractions along with the time intervals during which the abstractions are to be asserted. The abstractions (called *constraint templates*) are then used to create constraints at the appropriate times. For example, at times 2 and 3 the template $\forall t, x_t + y_t = z_t$ would create $x_2 + y_2 = z_2$ and $x_3 + y_3 = z_3$. In a Kaleidoscope'90 program, the keyword `always` creates a permanent constraint template, the keyword `once` creates a constraint template that is asserted for a single time interval, and the `while...assert` syntax creates a constraint template that is asserted for a dynamically determined duration. Figure 1 contains three example constraint templates.

In the imperative paradigm, the value of a variable (a location in the value store) does not change unless that variable is explicitly assigned to.[3] In the CIP paradigm this property is established by very weak "stay" constraints for each variable, i.e., $\forall x, \forall t, very\_weak\ x_t = x_{t-1}$. These stay constraints serve the role of the Frame Axioms used in AI problem solving systems. They are preferences rather than requirements: if an assignment or a stronger constraint has a different value for the variable, the stay constraint should be ignored.

[3]Except in the presence of aliasing or pointers—which is why these features make it harder to describe the semantics of real imperative languages.

Finally, to integrate destructive assignment with declarative constraints, we redefine assignment to be a *synchronic* equality constraint—a constraint between the values of variables in two different time intervals. The expression on the left side refers to the next time, and the expression on the right side refers to the current time. For example, `x:=x+y` becomes $x_{t+1} = x_t + y_t$. As a result, all of the computation in the data store involves constraints and no special cases are needed to accommodate assignment or other side effects. Since assignments are merely constraints, they may have complex expressions on both sides. (For example, `b*c-5 := d+sin(e)` becomes $b_{t+1} * c_{t+1} - 5 = d_t + \sin(e_t)$.)

Because all computation in a CIP program is done with constraints, the imperative actions of *writing* to the value store and *computing* a value are replaced by *adding* one or more constraints to the value store. The other operation on the value store is a *read*. In a CIP language, a read is necessary only when a value is being output to the external world or is needed to select which branch of a conditional to take. There may be many values for a variable that satisfy the constraints equally well. Since we wish to support an imperative programming style, we select one of the permissible values, rather than returning a set of all the permissible values of the variable, or backtracking through the possibilities, as in a logic program. (This choice is also more compatible with writing interactive graphical applications—a primary application area for CIP languages—as it would be difficult to display an icon on the screen at multiple locations all at the same time.) The selected value is then frozen so that subsequent reads of the same pellucid variable will return the same value. (Freezing the value is equivalent to adding a required constraint that the pellucid variable be equal to a constant, i.e., $x_t = c$.)

To prevent various paradoxes in which the future modifies the past, synchronic constraints are annotated with read-only annotations so that values can flow only forward in time. (See [3] for a formal definition of read-only annotations in constraint hierar-

chies.) Thus `x:=x+1` actually becomes $x_{t+1} = x_t? + 1$, where $x_t?$ denotes a read-only use of the pellucid variable x_t.

3.3 Objects

Modern programming languages support user-defined data types (often including abstract data types), and operations over those data types. Supporting user-defined data types in a general way presents a problem for constraint languages: user-defined data types do not always contain enough semantic information for the built-in constraint solvers to reason about constraints on them at their level of abstraction. Three approaches to this problem are:

1. Limit the kinds of constraints that can be applied to user defined data types, or the power of the solver for these constraints.

2. Allow the programmer to define new solvers for user-defined data types, either within the program itself or dynamically linked to the compiled program.

3. Provide a mechanism to define constraints over user-defined data types in terms of simpler constraints, ultimately reducing them to constraints over primitive data types. This approach, which we call *splitting*, is similar to how existing imperative languages deal with user defined data types: an imperative program ultimately defines operations over its user-defined data types in terms of operations over the primitive data types.

For example, consider a set of user-defined data types on geometric objects. We would like to define suitable constraints on these objects, e.g., point-on-line, parallel-lines, and so forth. However, a built-in constraint solver typically would not have enough information to reason about the geometric constraints at the geometric level. We could limit the kinds of user-definable constraints (e.g., to equality constraints only) or the power of the solver (e.g., to local propagation only). We could define a new solver, e.g., Kramer's solver for geometric constraint satisfaction [18], and link it to the existing solvers in some fashion. Finally, we could define the geometric constraints in terms of primitive constraints and data types, in this case algebraically using the real numbers.

In our current design the third approach (splitting) is used.[4] A *constraint constructor* is a procedure that defines how a particular constraint over a user-defined data type is to be split into constraints over its component parts. Those component constraints may be further split, and so on, until all the resulting constraints are over primitive types, and can be solved directly by the built-in constraint solvers.

Object-oriented languages add inheritance and dynamic binding of operation names to operations to the basic notion of abstract data types. To handle the dynamic binding of names to operations, when executed, a constraint statement in a CIP program creates a constraint template. The constraint template is bound to, and calls, one constraint constructor for each time interval. The constructor to which it is bound depends on the concrete type of the values in the pellucid variables it constrains. This dynamic binding supports object-oriented programming, and allows the concrete types of the constrained variables to change during the execution of the program. (If static binding were used, for example if the constraint template were bound to a constructor at creation time, then the concrete types of the variables would be fixed.)

$$\frac{\textit{Program}}{\text{statement}} \xrightarrow{\text{installs}} \frac{\textit{Runtime}}{\text{template}} \xrightarrow{\text{calls}} \frac{\textit{Program}}{\text{constructor}}$$

The constraint constructor either splits the constraint into further constraint templates or, at the lowest level, generates a primitive constraint. Constraint constructors are procedures and thus may contain any legal statement or code fragment including iteration, recursion, and assignment. Each constructor executes in a nested local time scope, so that time advances affect only local variables, but not non-local ones.

In classical object-oriented languages, such as Smalltalk-80, the operation chosen for a given name is determined by the concrete type of the first argument only. (In Smalltalk-80 terminology, the method invoked by a given message is determined only by the class of the receiver.) However, this scheme is inadequate for an object-oriented CIP language, since we want to support multi-directional constraints. For example, consider a three-argument *plus* constraint `p+q=r`. If `p` were known, then the appropriate *plus* constructor could be selected based on the concrete type of `p`. However, if `q` and `r` were known, and `p` were unknown, then this would fail. Hence we use *multi-methods*, in which the operation invoked is determined by the concrete types of all of the arguments rather

[4] In future CIP languages, we would retain splitting as the normal way of handling user-defined constraints. However, as Kramer's work demonstrates, solving the constraints at the appropriate level of abstraction adds considerable power; this argues for allowing special-purpose solvers to be employed as well.

Recursion

```
constructor =( t:Tree, v:Tree )
    always: t.value = v.value;
    always: t.left  = v.left;
    always: t.right = v.right;
end =;
constructor =( t:nil, v:nil )
end =;
```

Iteration

```
constructor sum( a:Array, s:Number )
    var i, partial : Number;
    partial := 0;
    for i := a.first to a.last do
        partial := partial + a[i];
    end for;
    always: s = partial;
end sum;
```

Figure 2: Constraint Constructor Examples

than just the first. (Multi-methods were first used in the CLOS extension to COMMON LISP [1].)

Figure 2 contains two example constraint constructors, one that recursively defines equality for trees, and another that iteratively defines the "sum" constraint for arrays.

4 A Prototype Implementation

In [11] we describe the design and implementation of a prototype constraint imperative language, Kaleidoscope'90. Kaleidoscope'90 is an integration of a typed dialect of Smalltalk-80 [17] and an enhanced version of the constraint system from ThingLab II [22]. This initial implementation interprets the constraint imperative programming semantics directly, resulting in a robust but very slow implementation.

The interpreter is divided into three sections: a pre-compiler, an imperative engine, and a constraint-based value store. The pre-compiler converts the Kaleidoscope'90 source program into a K-code object program. There are six basic K-codes: *add* a constraint template to the active set, *remove* a constraint template from the active set, *read* a value from the value store, *call* a procedure, *branch* conditionally or unconditionally, and *advance* time. When the advance time K-code is executed (typically at the end of each source statement), each constraint template in the active set is transmitted to the constraint-based value store. Thus, a `once` constraint statement corresponds to a template that is added to, and then promptly removed from, the active set, whereas an `always` statement corresponds to a template that is added and never removed.

The constraint-based value store contains two comingled hyper-graphs: a primitive one and a compound one. The primitive graph contains a fixed class of constraints over the primitive domains (floating point numbers and booleans). The compound graph contains a variety of hyper-edges which, together, support constraint templates over objects. As described earlier, the primitive constraints cannot be directly created by the program—instead, the program creates constraint templates which, in turn, create hyper-edges in the compound graph. These in turn are bound to the constraint constructors that create the primitive constraints. Thus, the process for determining a pellucid variable's value is first to reduce the compound graph to a primitive graph, and then to use the built-in primitive constraint solver to solve the primitive graph. The compound graph is reduced as follows:

1. Type constraints are solved to determine concrete types for the pellucid variables.
2. Constraint constructors are bound and called, based on the concrete types.
3. Constraint constructors on whole objects are executed before those on their component parts to ensure that all primitive constraints on an object are considered, including those created by constraint constructors executed on any enclosing objects.

The primitive constraint solver uses local propagation when possible. When local propagation fails, if the remaining constraints are linear equalities or inequalities over the real numbers, the solver can resort to a variant of the Simplex method adapted for constraint hierarchies.

Although the goal of this initial implementation was a proof of concept rather than an efficient implementation, we did add a few optimizations to improve its performance. One technique is to replace equality constraints by identity constraints when possible. (We can't always perform this replacement, since in an object-oriented language the programmer is allowed to redefine equality between objects.) When this substitution is used, it reduces the number of objects, pellucid variables, constraint templates, and primitive

constraints that are created, and thus diminishes the overhead of reducing the compound graph and solving the primitive constraints.

Another technique we used is to combine multiple smaller constraint templates into a single larger, amalgamated template. Our motivation is that solving a large number of constraints (even simple ones) is expensive. Thus, the strict object-oriented scheme of decomposing a single constraint expression into a multitude of smaller constraint expressions, each containing a single operator, substantially increases the number of edges in the compound graph. This is obviously undesirable. Note that this amalgamation technique is similar to using a graph preprocessor to manipulate and improve the compound graph before the actual constraint solver is invoked.

These two simple techniques—replacing equality by identity, and amalgamating constraints—more than doubled the speed of our prototype implementation, and have supported our belief that better performance from our second generation implementation will be possible.

5 Related Work

We can roughly divide the related work on constraints into research on using constraints in applications, and on embedding constraints in programming languages. Applications have included interactive graphics, layout systems, user interface construction systems, various artificial intelligence systems, and design, analysis, and simulation systems. See [9, 11, 21] for citations and a comparative discussion of these applications. (Due to space limitations we don't describe them here.)

In the programming language arena, one of the earliest efforts was that of Steele [25], whose dissertation describes work on a general-purpose constraint language using local propagation to find solutions. Subsequently, Leler [21] designed and implemented a constraint language based on augmented term rewriting. Lamport and Schneider [20] propose adding constraints to an imperative language, as a uniform approach encompassing both aliasing and typing. Their primary motivation is the development of proof systems. Perhaps the closest work to that described here is Horn's Siri language [14, 15], which is also a hybrid object-oriented constraint imperative language. There are a number of differences between Siri and CIP. First, Siri uses only required constraints, rather than a constraint hierarchy. Therefore, the user must explicitly indicate when values are to remain the same as time advances. Second, Siri uses term rewriting (as in Leler's work) as its constraint satisfaction mechanism rather than DeltaBlue and DeltaStar. Third, Siri uses a single abstraction mechanism, a *constraint pattern*, for object description, modification, and evaluation, rather than more standard approach taken here of using separate mechanisms for these tasks. (This uniform use of patterns is analogous to their use in the BETA language [19].)

Most of the current activity in constraint languages is based on logic programming. Jaffar and Lassez have defined a general scheme, Constraint Logic Programming (CLP), for integrating constraints with logic programming [4, 16]. A number of instances of this scheme have now been implemented, including CLP($\mathcal{R}$) [12], Prolog III [5], and CHIP [6, 26]. Two generalizations of the CLP scheme are CLP* [13], which generalizes CLP by allowing predicates to be defined dynamically as first class objects, and HCLP [2, 28], which generalizes CLP by including constraint hierarchies rather than just required constraints. Finally, in the cc languages [23, 24] the conventional store of a Von Neumann computer is replaced by one that holds constraints. Concurrently executing agents communicate by *asking* and *telling* constraints to this store.

6 Future Work

We are currently redesigning and simplifying our language to produce Kaleidoscope'91. In addition, we are designing a more efficient implementation, which will employ a mixed interpretation/compilation strategy. Because constraints can be added and removed dynamically in a CIP program, in general it isn't possible to remove all runtime invocations of the constraint solver, but it is frequently possible to eliminate many of them. An obvious example of this optimization is assignment to an otherwise unconstrained variable: a simple LOAD-STORE instruction pair is a better choice than constructing a full compound graph, reducing it, and then solving the resulting equality constraint. Chapter 7 of [11] discusses other compilation opportunities.

Our goal is to provide an implementation of the general model (i.e., CIP) in such a way that any given program pays the cost of only the features that it uses. Once this new implementation is operational, we plan to further test the expressive power of constraint imperative programming by using it to write selected non-trivial programs. The experience we gain from this endeavor will then feed back to additional improvements in the design and implementation of Kaleidoscope. Finally, we plan to investigate environmental support for CIP languages.

Acknowledgements

Thanks to Kirsten Freeman-Benson, Bruce Horn, Leslie Lamport, Michael Sannella, Joe Sherman, Kevin Sullivan, Bill Wadge, and Molly Wilson for comments on drafts of this paper. This work was supported in part by the University of Victoria, by the National Science Foundation under Grant No. IRI-9102938, by a Graduate Fellowship from the National Science Foundation for Bjorn Freeman-Benson, and by a gift from Apple Computer.

References

[1] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System Specification X3J13. *SIGPLAN Notices*, September 1988.

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

[3] Alan Borning, Molly Wilson, and Bjorn Freeman-Benson. Read-Only Annotations in Constraint Hierarchies. Technical Report 91-07-04, Department of Computer Science and Engineering, University of Washington, July 1991.

[4] Jacques Cohen. Constraint Logic Programming Languages. *Communications of the ACM*, pages 52–68, July 1990.

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

[6] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Bertheir. The Constraint Logic Programming Language CHIP. In *Proceedings Fifth Generation Computer Systems-88*, 1988.

[7] Bjorn Freeman-Benson. Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming. In *Proceedings of the 1990 Conference on Object-Oriented Programming Systems, Languages, and Applications, and European Conference on Object-Oriented Programming*, pages 77–88, Ottawa, Canada, October 1990. ACM.

[8] Bjorn Freeman-Benson and Alan Borning. Integrating Constraints with an Object-Oriented Language. In *Proceedings of the 1992 European Conference on Object-Oriented Languages*, June 1992. To appear.

[9] Bjorn Freeman-Benson, John Maloney, and Alan Borning. An Incremental Constraint Solver. *Communications of the ACM*, 33(1):54–63, January 1990.

[10] Bjorn Freeman-Benson and Molly Wilson. DeltaStar, How I Wonder What You Are: A General Algorithm for Incremental Satisfaction of Constraint Hierarchies. Technical Report 90-05-02, Department of Computer Science and Engineering, University of Washington, May 1990.

[11] Bjorn N. 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.

[12] Nevin Heintze, Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. The CLP($\mathcal{R}$) Programmer's Manual Version 1.1. Technical report, IBM T.J. Watson Research Center, November 1991.

[13] Timothy J. Hickey. CLP* and Constraint Abstraction. In *Proceedings of the Sixteenth Annual Principles of Programming Languages Symposium*, pages 125–133, Austin, Texas, January 1989. ACM.

[14] Bruce Horn. Thesis Proposal: A Constrained-Object Language for Reactive Program Implementation. Carnegie Mellon University, May 1990.

[15] Bruce Horn. Properties of User Interface Systems and the Siri Programming Language. In Brad Myers, editor, *Languages for Developing User Interfaces*. Jones and Bartlett, Boston, 1992.

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

[17] Ralph E. Johnson. Type-Checking Smalltalk. In *Proceedings of the 1986 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 315–321, Portland, Oregon, November 1986. ACM.

[18] Glenn Kramer, Jahir Pabon, Walid Keirouz, and Robert Young. Geometric Constraint Satisfaction Problems. In *Working Notes of the AAAI Spring Symposium on Constraint-Based Reasoning*, pages 242–251, Stanford, March 1991.

[19] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pederson, and Kirsten Nygaard. Abstraction Mechanisms in the BETA Programming Language. In *Proceedings of the Tenth Annual Principles of Programming Languages Symposium*, Austin, Texas, January 1983. ACM.

[20] Leslie Lamport and Fred B. Schneider. Constraints: A Uniform Approach to Aliasing and Typing. In *Proceedings of the Twelfth Annual Principles of Programming Languages Symposium*, pages 205–216, New Orleans, Louisiana, January 1985. ACM.

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

[22] John Maloney, Alan Borning, and Bjorn Freeman-Benson. Constraint Technology for User-Interface Construction in ThingLab II. In *Proceedings of the 1989 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, New Orleans, October 1989. ACM.

[23] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of the 18th Annual Principles of Programming Languages Symposium*. ACM, 1991.

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

[25] Guy L. Steele. *The Definition and Implementation of a Computer Programming Language Based on Constraints*. PhD thesis, MIT, August 1980. Published as MIT-AI TR 595, August 1980.

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

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

[28] Molly Wilson and Alan Borning. Extending Hierarchical Constraint Logic Programming: Nonmonotonicity and Inter-Hierarchy Comparison. In *Proceedings of the North American Conference on Logic Programming*, Cleveland, October 1989.