Integrating Pointer Variables into One-Way Constraint Models

Brad Vander Zanden

Computer Science Department
University of Tennessee
Knoxville, TN 37996
bvz@cs.utk.edu

Brad A. Myers
Dario A. Giuse

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
brad.myers@cs.cmu.edu
dzg@cs.cmu.edu

Pedro Szekely

USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
szekely@isi.edu

Abstract

Pointer variables have long been considered useful for constructing and manipulating data structures in traditional programming languages. This paper discusses how pointer variables can be integrated into one-way constraint models and indicates how these constraints can be usefully employed in user interfaces. Pointer variables allow constraints to model a wide array of dynamic application behavior, simplify the implementation of structured objects and demonstrational systems, and improve the storage and efficiency of constraint-based applications. This paper also presents two incremental algorithms—one lazy and one eager—for solving constraints with pointer variables. Both algorithms are capable of handling 1) arbitrary systems of one-way constraints, including constraints that involve cycles; and 2) editing models that allow multiple changes between calls to the constraint solver. These algorithms are also fault-tolerant in that they can gracefully handle and recover from formulas that crash due to programmer error. Constraints that use pointer variables have been implemented in a comprehensive user interface toolkit, Garnet, and our experiences with applications written in Garnet have proven the usefulness of pointer-variable constraints. Many large scale applications have now been implemented using these constraints.

Categories and Subject Descriptors: D.2.2 [Software Engineering]: Tools and Techniques—User Interfaces; D.2.3 [Software Engineering]: Coding—Program editors; D.2.6 [Software Engineering]: Programming Environments; I.1.2 [Computing Methodologies]: Algorithms—Nonalgebraic Algorithms; I.1.3 [Computing Methodologies]: Languages and Systems—Evaluation Strategies

General Terms: Algorithms, Design, Languages

Additional Keywords and Phrases: Constraints, development tools, incremental algorithms, Garnet

1 Introduction

Many tools, including user interface toolkits [33, 13, 3, 5, 14], programming environments [40], and picture drawing tools [49, 39, 26, 2], are increasingly integrating constraints into their programming models. A constraint denotes a relationship among two or more objects. For example, a designer might write the following equation to position a circle 10 pixels to the right of a rectangle:

```
circle.left = rect34.right + 10
```

A constraint satisfier automatically maintains these relationships and ensures that changed data is propagated to the appropriate places. Thus, if a user drags the rectangle around the screen, a constraint solver continuously resatisfies the above equation, causing the circle to follow the rectangle around the display.

The most widely used types of constraints in user interfaces are one-way constraints. A one-way constraint has a method associated with it that contains a set of input variables, an output variable, and an arbitrary piece of code that computes the output variable based on the values of the input variables. The advantages of one-way constraints are 1) there are efficient, incremental algorithms for satisfying them; 2) their consequences are easily understood by users; and 3) they are domain-independent.

However, the usefulness of the conventional, one-way constraint model is restricted since it does not support pointer variables. Instead, references to objects must be hardcoded into a constraint, as `rect34` is in the above example. This restriction makes it difficult to use constraints with data structures that are typically implemented using pointers, such as lists, trees, and graphs. For example, a designer of a list cannot write constraints that depend on the previous element in the list, because insertions and deletions can cause that element to change. A second drawback of the conventional model is that constraints can only reference a fixed set of objects. Thus, a designer cannot readily write constraints that apply to dynamically changing sets of objects, such as the neighbors of a node in a graph or the set of objects that a feedback object might highlight in a graphical interface.

Both these shortcomings can be remedied by adding pointer variables to constraints that allow objects to reference other objects indirectly. These constraints are called *indirect reference constraints*. For example, the above constraint could be rewritten as:

```
circle.left = self.object_over.right + 10
circle.object_over = rect34
```

where `self` refers to the object containing the variable `left`, in this case, the circle. The constraint will be satisfied by following the link in `object_over` to `rect34`, retrieving the value of `rect34`'s `right` instance variable, and adding 10 to it. The result is assigned to `circle.left`. By changing the value of the variable `object_over`, the circle can be positioned to the right of any object.

In our own research with graphical interfaces, we have found that indirect reference constraints greatly simplify the implementation of many interface features and enable the implementation of new ones that otherwise would have been unwieldy:

- Feedback, in which objects, such as checkmarks or inverted rectangles, may appear with any item in a set of objects;
- Prototype-Instance models, in which constraints are inherited from prototypes, and references in the inherited constraints must be adjusted so that they are relative to the instance rather than the prototype;
- Programming by example, in which constraints that are demonstrated for example objects must be generalized to work with any objects;
- Abstract specification of layouts, in which generic objects are laid out using constraints, and the specific objects are filled in later, based on such parameters as the availabilty of screen space;

- Animations, in which objects are frequently constrained to new and different objects, for example objects moving between the machines on a factory floor.

However, the usefulness of pointer-variable based constraints extends well beyond the realm of graphical interfaces. There are many invariant relationships in commonly used algorithms that could be usefully maintained by one-way constraints. For example, the value of a node in a critical path computation should always be the maximum of its neighbors plus the values of their edges, the interior nodes of a 2-3 tree must keep track of the largest leaves in their subtrees, and dynamic programming algorithms must build up partial values from neighbor values. Constraints can also be used to communicate information between multiple threads of a dialog, to compute the attribute values of objects, and to monitor the states of various objects.

In order to make constraint satisfaction fast enough to provide interactive feedback, we have extended lazy and eager evaluation algorithms for one-way constraints to handle pointer variables and dynamically changing sets of objects. The lazy evaluator presented in this paper uses a nullification/reevaluation scheme and can incorporate Hudson's optimal lazy evaluation algorithm [22] for direct reference constraints. The eager evaluator presented in this paper assigns position numbers to variables based on their position in topological order and evaluates constraints in the order indicated by these position numbers. This eager algorithm uses a variation of an eager evaluator presented by Hoover [16]. Both of the algorithms presented in this paper can handle arbitrary systems of one-way constraints, including systems that contain cycles. They can also handle multiple edits to the constraint system (e.g., adding constraints, deleting constraints, or changing the values of variables) between successive calls to the constraint solver. Finally, they are able to handle and correctly continue from formula crashes due to programmer error. This capability is important in interpreted environments, such as programming environments, debugging environments, and spreadsheet environments, where the user is given the opportunity to correct an error and continue.

Indirect references were the key extension to constraints which allowed Garnet to be the first comprehensive user interface toolkit to be built on top of the constraint system [33, 35]. This includes the graphical object system, the handling of the input, all the widget libraries, and the higher-level interactive tools. For example, a Garnet text button widget contains 43 constraints internally and the Lapidary graphical interface builder contains 16,700 constraints [36]. In Garnet, constraints are used instead of methods to implement many types of behavior. For example, some of their uses in the text button widget include: 1) aligning the parts of the text button, such as the label and button; 2) controlling the behavior of the text buttons, such as whether they are active or inactive and whether the buttons are toggled or simply set; 3) communicating information between the input handlers and the graphical parts of the text buttons so that the appropriate graphical feedback is provided; and 4) computing the value that the widget passes to the application. This emphasis on constraints helps define a new style of programming, one in which the focus is on computing data values instead of writing methods [36]. The desirability of this approach has been validated in practice. Garnet has over 100 users in over 60 projects who have used indirect reference constraints to generate numerous applications. Many of these applications contain thousands of indirect reference constraints. The success of indirect constraints in Garnet has inspired their use in many other systems including MultiGarnet [43], Rendezvous [15], and Eval/vite [23].

The rest of this paper is organized as follows. The next section describes how a number of user interface applications can be implemented using indirect reference constraints. Section 3 discusses how indirect reference constraints can enhance the performance of an application and decrease its storage demands, while section 4 presents incremental algorithms for solving indirect reference constraints. Section 5 describes our implementation experiences with pointer variables and their performance in practice. Finally Section 6 describes related work, and Sections 7 and 8 presents our ideas for future research and conclusions.

2 Example Applications of Indirect Reference Constraints

Indirect reference constraints can be used to implement many parts of an application that are difficult or infeasible to implement with direct reference constraints. Since our experience is in graphical interfaces, many of our examples are drawn from graphical applications and include feedback, copying and instancing of composite objects with constraints in them, programming by example, and animations. However, we also discuss how indirect reference constraints could be useful in other applications, such as list, graph, and tree algorithms, and in process monitoring. Yet other applications that are not discussed might include semantic checking in programming environments, timing analysis in computer-aided design systems, and modeling of various objects and processes in simulation systems.

2.1 Tree, Graph, and List Algorithms

Many algorithms that work on trees, graphs, and lists maintain invariants that must be updated each time the data structure changes. Several examples were given in the introduction, such as computing the critical path in a directed graph or keeping track of the largest leaves in a 2-3 tree. These relationships can be naturally maintained using one-way constraints. For example, any time a new leaf is added to a 2-3 tree, constraints can automatically update the largest leaf information in the nodes on the path leading from the inserted leaf to the root. Similarly, when a new node is added to a directed graph, the constraints in the nodes downstream of the inserted node can automatically recompute the critical path. Since these data structures are often implemented using pointers, it considerably simplifies the implementation of the constraints if the constraints can reference other elements of the data structures through pointers. For example, a constraint computing the critical path to a node might be written as:

```
node.critical_path =  begin
                      max = -1
                      for each edge ∈ self.neighbors do
                         max = max(max,edge.vertex.critical_path + edge.cost)
                      return(max)
                      end
```

It would be very inconvenient to use direct reference constraints, because the application would have to delete the old constraints and insert new constraints each time elements were added to or deleted from the data structures. It could also be quite expensive, since the initialization and deletion of constraints can involve considerable overhead (see Section 3 for a further discussion of this issue).

2.2 Monitors

A useful purpose of constraints is to serve as monitors that trigger certain actions when data values change or exceed certain limits. For example, constraints might be used to monitor objects moving about on a factory floor, packets moving through a network, or data in a program. The constraints might be associated with a machine, a network site, or a data structure. Thus the objects moving through the sites would continuously change, but there would be no need for the constraints to change. It would be easiest to have the constraints reference the objects through pointers, and simply reset the pointers each time a new object arrived. For example, a machine might check the width of a part to make sure it meets certain tolerances before sending it on. The constraint might take the form:

```
tolerance_check = if part.width < min_tolerance or
                           part.width > max_tolerance
                  then
                        call routine to take corrective action
```

2.3 Feedback

Most direct manipulation interfaces provide feedback to the user while performing an operation. For example, a rectangle may surround the item that the user is currently pointing at in a menu (Figure 1.a). While it is generally impractical to handle feedback objects using direct reference constraints, they are easily handled using indirect reference constraints. For example, direct reference constraints will allow a rectangle to highlight only one of the items in Figure 1.a. In contrast, indirect reference constraints allow the feedback object to reference any of these menu items through a variable, such as `object_over`. This technique works equally well for feedback objects that highlight a fixed set of objects, such as the objects in a menu, or a dynamic set of objects, such as the objects in a drawing window (Figures 1.a and 1.b). Indirect reference constraints can also be used to control the visibility of a feedback object:

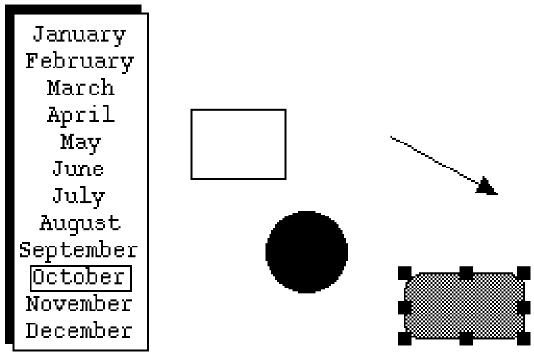
```
feedback.visible = if self.object_over == nil
                        then false else true
```

If none of the menu items is selected, the `object_over` variable is nil, and the feedback object is invisible; otherwise the `object_over` variable points to the selected item and the feedback object highlights this item.

2.4 Structured Objects

Pointer variables simplify the integration of constraints into a structured object system. A structured object consists of several parts, such as the labeled box in Figure 2.a, which consists of a rectangle and a piece of text. Typically these parts are mutually constrained. For example, the label is centered inside the box and the size of the box depends on the size of the label.

Interactive applications need to make copies or instances of these objects at runtime (e.g., creating new objects in a drawing program, creating new circuit elements in a circuit simulation program). These operations can be easily implemented using indirect reference constraints, but are more difficult to implement in regular constraint systems.



(a) (b)

Figure 1: The rectangular feedback object in the menu and the selection handles in the drawing editor use constraints to center themselves over the selected items and to change their dimensions to the dimensions of the selected item. By indirectly accessing a selected item through the variable `object_over`, the feedback objects are able to appear both over any item in a static set of objects, such as the menu items (a), or any item in a dynamic set of objects, such as the objects in the drawing editor (b).

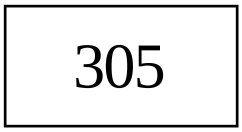
In an indirect reference constraint system, each object maintains a pointer to its parent, and a set of pointers to its children (Figure 2.b). Constraints reference objects by following the appropriate pointers. For example, if the label's parent pointer is contained in the variable `parent` and the labeled box keeps pointers to its children in the variables `label` and `box`, then the label can be centered inside the box using the following constraints:

```
label.center_x = self.parent.box.center_x
label.center_y = self.parent.box.center_y
```

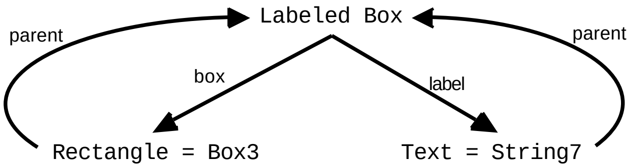
To create an instance of an object, the object system creates instances of each of the object's components and sets the pointer variables (Figure 2.c). The object system also creates instances of each of the constraints in the prototype's components and stores them in the appropriate places in the new instance's components. No changes are needed to the constraint expression. The constraints in the newly created objects will automatically reference the appropriate objects since they will follow the pointers in the instance objects rather than in the prototype objects. For example, the constraint that computes the value for `center_x` in the label instance will follow the `parent` and `box` pointers in the labeled box structure hierarchy and retrieve the `center_x` value of the rectangle instance.

This system of pointers and indirect references also makes it easy to modify the parts of a structured object. For example, suppose constraints control the layout of the binary tree in Figure 3.a and that an application swaps two subtrees as shown in Figure 3.b[1] The swap command can be implemented

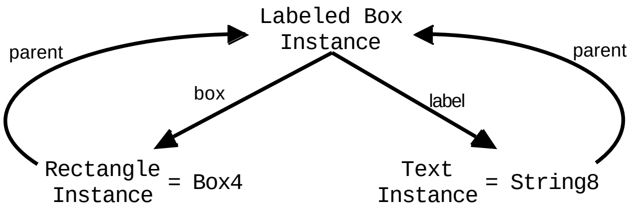
[1]The layout rules for the binary trees can be found in [50, pp. 286-287].



(a)



(b)

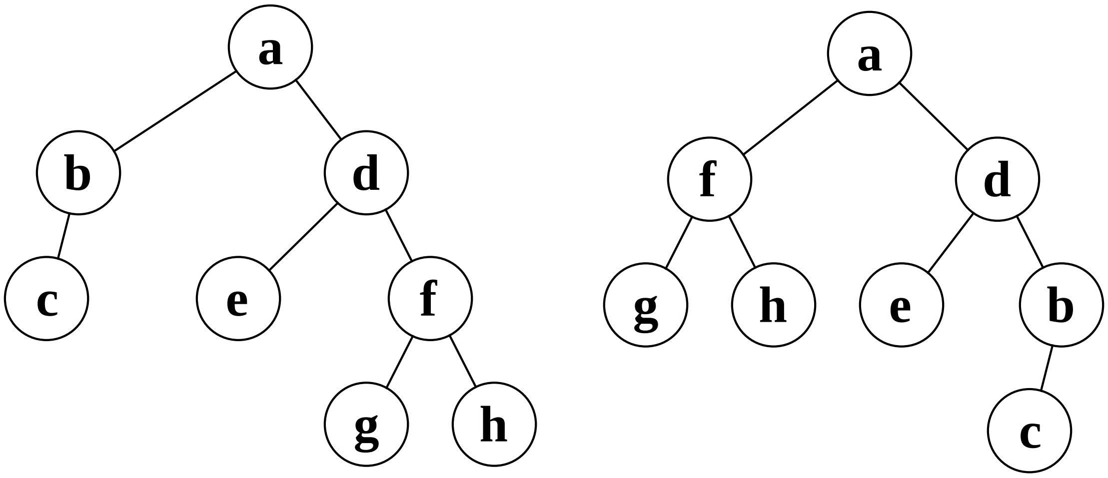


(c)

Figure 2: Structured objects, such as this labeled box (a), are built up from other objects, such as a rectangle and some text (b). Each object maintains pointers to its parent and its children, so that constraints can indirectly reference objects through pointers. This facilitates the copying and instancing of objects, since the object system simply sets the pointers in the new objects, and the constraints automatically reference the appropriate objects (c).

by changing the parent pointers of the swapped subtrees and updating the children pointers of the subtrees' old and new parents.

A direct reference system could conceivably support the instancing of objects through ''creation-time'' indirect reference constraints. In this scheme, the user would supply one or more paths with the constraint prototype that could be resolved into the appropriate references when an instance of the constraint was created at run-time (a path would be a set of pointers, such as `self.parent.box` that could be used to traverse the object hierarchy). This approach is used, for example, in ThingLab [4]. However, dynamic modifications to the objects' structure cannot be directly handled in this scheme because the objects referenced by the paths are not allowed to change dynamically.



(a) (b)

Figure 3: (a) An application that visualizes binary trees; (b) a modified binary tree with the subtree rooted at `b` swapped with the subtree rooted at `f`.

2.5 Programming by Example

Indirect reference constraints make it easier to implement systems that employ demonstrational programming [37], such as the graphical interactive design tool Lapidary [32]. In a demonstrational system, a user draws an example picture or demonstrates an example behavior, and then the system creates a generalized prototype object or behavior by figuring out which values in the picture or behavior should be parameters. If the demonstrational system uses indirect reference constraints, then it is easy to generalize these examples. In fact, the example that the user draws or demonstrates is already a prototype, since the object can be instanced or copied using the scheme described in the previous section.

In Figure 4, a designer is using Lapidary to create a boxes-and-arrows editor. The designer has drawn an example picture in which the arrows are attached to the center of the boxes that they connect. Lapidary represents the constraints of the line internally as indirect reference constraints:

```
arrow.endpt1 = self.from_obj.center
arrow.endpt2 = self.to_obj.center
arrow.from_obj = box1
arrow.to_obj = box2
```

The designer can save this arrow and the application can use it as a prototype. When the boxes-and-arrows editor creates instances of this arrow, it stores pointers to the appropriate boxes in the `from_obj` and `to_obj` variables, and the constraints automatically attach the endpoints of the arrow instance to the centers of the boxes. The application does not need to know anything about the

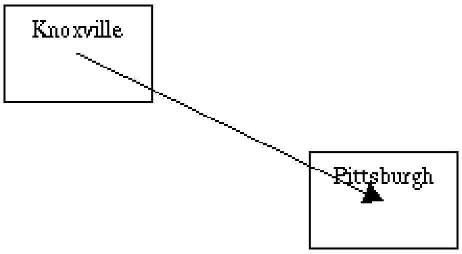


Figure 4: An example picture demonstrating that the endpoints of an arrow should be attached to the centers of the boxes it connects. An application builder will generalize this arrow into a prototype that can connect any pair of boxes.

constraints, structure, or graphics of the line. The constraints on the endpoints could connect centers to centers, right sides to left sides, or even use a complex formula that computes the nearest sides and tries to avoid crossing other lines.

2.6 Animations

Animations often require objects to move smoothly between various points of the display. For example, sort animations show objects moving around in linked lists or arrays, navigation systems move objects around transportation corridors, and manufacturing systems route objects through the machines on a factory floor. Indirect reference constraints model this motion by using variables to reference the beginning and target positions.

For example, suppose we want the carton in Figure 5 to glide from station A to station B as if it were on a conveyor belt. This could be done by writing a set of constraints that interpolate the carton's position based on a percentage and the stations' positions:

```
carton.percent = 0
carton.x_distance = self.to_station.center_x - self.from_station.center_x
carton.center_x = self.from_station.center_x
                       + (self.x_distance * self.percent)
carton.to_station = station_b
carton.from_station = station_a
```

`x_distance` computes the distance between the old and new stations, and `center_x` contains the current x position of the carton. As the application program increments `carton.percent` from 0 to 1, the carton moves smoothly between its old and new assembly station. To prepare for the next step of the animation, the application resets the percentage to 0, stores station B in `from_station`, and stores a new station, C, in `to_station`.

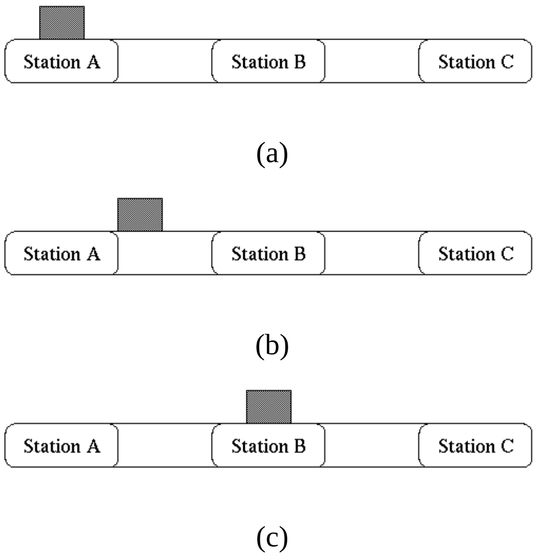


Figure 5: An assembly line with stations connected by a conveyor belt. A carton should be centered above the station that is currently processing it (a), and cartons should move smoothly from one station to the next, (b) and (c).

3 Performance Implications for Applications

The generalization of constraints using pointer variables can improve the efficiency of an application by decreasing the number of constraints and objects it uses, decreasing the size of the constraints it uses, and decreasing the number of constraints that it must dynamically create and delete. Indirect reference constraints also make it easier for the constraint system to maintain only one rather than multiple copies of a constraint, and make it easier to statically compile the constraints. The use of pointer variables does not preclude the use of other optimization techniques, such as constant propagation and compiling constraints into modules of constraints [9], although as in compiler optimization, pointer variables can make this analysis more difficult.

Storage improvements come in two forms. First, by allowing objects to be constrained to many different objects, indirect reference constraints may significantly decrease the number of objects which an application creates. For example, suppose a feedback object should highlight the currently selected item in a menu, as in Figure 1.a. If direct reference constraints are the only constraints available, the designer may have to create a separate feedback object for *each* menu item, since the constraints will bind each feedback object to exactly one item. Thus 12 feedback objects, each with five constraints will be required. However, as noted in Section 2.3, indirect reference constraints allow a feedback object to highlight any menu item, and thus one feedback object suffices. Second, indirect reference constraints can be written much more compactly and elegantly than direct reference constraints. Returning to the feedback example, a clever designer who is working with direct reference constraints might be able to use only one feedback object by defining constraints which reference every object in a menu and which describe how the feedback object should highlight each menu item. For example, to implement the feedback object in Figure 1.a, the designer might write the following constraint to define the left side of the feedback object:

```
feedback.left = case month
                  "January": Jan.left
                  "February": Feb.left
                        ...
                  "December": Dec.left
```

where month is a string variable containing the currently selected month.

However, this solution has four drawbacks:

- Non-modular and inelegant: The constraint must be modified if the designer adds a new menu item.
- Space: The constraint must have twelve separate conditions and actions, which causes the code to occupy a considerable amount of space at runtime. Also, 12 dependency pointers, one for each object, must be maintained by the constraint system (a dependency pointer indicates that a constraint depends on an object).
- Efficiency: The constraint depends on all twelve objects. If one object changes, even if it is not the currently selected object, the constraint must be reevaluated.
- Dynamic Sets: The constraint only works for static sets of objects, since the objects must be hardcoded in the constraint. It cannot be used to describe dynamic sets of objects, such as the objects in the drawing window in Figure 1.b.

Indirect reference constraints suffer from none of these disadvantages. The corresponding indirect reference constraint would be:

```
feedback.left = self.object_over.left
```

where `object_over` is a pointer to the selected menu item. This constraint is both compact and modular. It occupies less space than the corresponding direct reference constraint and requires only two dependency pointers, one to the variable `object_over` and one to the selected menu item. It depends only on the `object_over` variable and the currently selected menu item, so it will only be reevaluated when absolutely necessary (i.e., when either the left of the selected menu item changes or a new menu item is selected). Finally, this type of constraint can handle dynamic sets of objects. If additional items are added to the menu, the constraint automatically deals with them without having to be rewritten.

The efficiency advantage of indirect reference constraints derives in part from their storage advantage. Fewer constraints means fewer constraints to solve, and thus, less work for an equation solver. For example, a constraint solver that employs eager evaluation might take only one fifth the time to set up the feedback for a five item menu using indirect reference constraints instead of direct reference constraints, because there are only one fifth as many constraints to solve (a lazy evaluator would require the same amount of time in either case because it would only evaluate the constraints associated with the feedback object which should be displayed).

Efficiency advantages also arise because fewer constraints have to be dynamically created and destroyed. To illustrate the reduction in dynamically created and destroyed constraints, consider the construction of a menu using direct reference constraints. It may be impractical from a storage standpoint to maintain one feedback object for each item. Thus, the application may maintain only one feedback object and destroy the old constraints and create new constraints each time the feedback moves to a new item. The overhead involved in destroying and creating these constraints can be avoided if indirect reference constraints are used. In Garnet, an indirect reference requires 68 microseconds more than a direct reference on an HP720 workstation running Lucid Common Lisp. Caching strategies discussed in Section 4.5 can reduce this additional time to essentially zero unless the pointer actually changes. Changing a pointer requires 26 microseconds. In contrast, destroying an old constraint and creating a new one requires 102 microseconds. Although these times are not individually significant, they can add up if several constraints must be created and destroyed on each mouse movement, and could reduce the interactive performance of an application.

Indirect reference constraints also simplify the construction of the constraint system. First, the formula for an indirect reference constraint can be stored in a prototype and instances of the prototype can maintain pointers to this formula. Thus, many instances of a prototype constraint can be created, but the formula is created only once (storing only one copy of the formula's code also decreases the amount of storage that constraints require). Second, indirect reference constraints can be statically compiled because the only parameter that has to be passed to the constraint is ‘‘self’’. This considerably simplifies implementing a constraint system in an existing general-purpose language. For example, Garnet constraints can be arbitrary Lisp code. Direct reference systems typically also maintain only one copy of a constraint and statically compile it. However, to accomplish this, they require the user to write the constraint as a function with parameters denoting the direct references, or else parse the constraint to determine the direct references. However, we have discovered that users find it irritating and cumbersome to have to define parameters for constraints. For example, it is much more elegant and compact to write

```
feedback.left = self.object_over.left
```

than to write

```
constraint left_align (obj) { obj.left }
feedback.left = left_align(self.object_over)
```

Similarly, it can be quite difficult to write a parser to search through each constraint and locate the direct references. Note that it may seem advantageous to force the user to declare pointers as parameters, so that various optimizations can be made during constraint solving (e.g., caching the values of pointer paths, so that the whole path does not have to be traversed each time, unless one of the pointers has changed—a pointer path is the portion of a reference before the last period, for example, `self.object_over` in the reference `self.object_over.left`). However, in practice, it is sufficient to assume that all but the last variable in a path is a pointer; thus declaring pointers provides no real advantage.

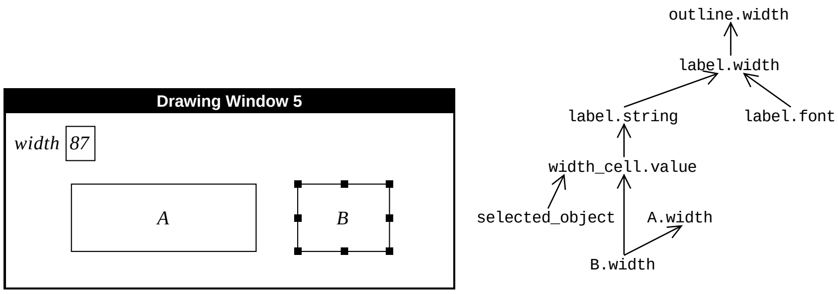
4 Implementation

So far we have discussed the uses and benefits of indirect reference constraints. In this section we describe how to implement indirect reference constraints. This section presents three algorithms for implementing indirect reference constraints—a lazy evaluation algorithm that works for both acyclic and cyclic graphs, an eager evaluation algorithm that works with acyclic graphs, and an eager evaluation algorithm that works with cyclic graphs. An introductory subsection describes the notation common to all three algorithms. Then each algorithm is described in a separate subsection with a high-level overview, the algorithm itself, and a pictorial example.

4.1 Notation

The fundamental data structure used by the evaluation algorithms is a directed dependency graph in which the nodes represent variables and the edges denote dependencies between variables. An edge directed from node `i` to node `j` indicates that the formula that computes the value of `j` uses the value of `i`. As an example, consider the simple interface in Figure 6.a. The cell labeled `width` displays the width of the selected object. The currently selected object is stored in the variable `selected_object`. The width cell consists of two objects—a text label and an outline rectangle. The constraints in this interface might consist of the following (the constraints that position the label ‘‘width’’ next to the width cell are not shown since they would reference a different set of variables than the ones shown below and thus would form an independent dependency graph):

```
width_cell.value = selected_object.width
width_cell.label.string = integer_to_string(self.parent.value)
width_cell.label.width = string_width(self.string, self.font)
width_cell.outline.width = min(50, self.parent.label.width + 10)
A.width = 2 * B.width
selected_object = B
```



(a) (b)

Figure 6: (a) An interface in which the width of the selected object is displayed in the cell labeled ‘‘width’’. (b) The dependency graph representing the constraints between the variables in this interface. Edges are directed from the variables that are requested to the variables that request them (i.e., the edges are dataflow edges). For example, the value that the width cell displays depends on the `selected_object` and the width of the selected object. In this case the selected object is B, so `width_cell.value` has incoming edges from `selected_object` and `B.width`.

The dependency graph for this set of constraints is shown in Figure 6.b. The label’s `width` attribute depends on the values of both the label’s `string` and `font` attributes, and thus there are directed edges from these two attributes to label’s `width` attribute. Similarly, width_cell’s `value` depends on `selected_object`, and through `selected_object` on `B.width`, so there are directed edges from these two variables to width_cell’s `value` attribute. The algorithms that maintain the dependency graph during constraint satisfaction can determine if a dependency exists between two variables by calling the function `get_dependency`. `get_dependency` takes two variables `x` and `y`, and returns a pointer to the data structure representing the dependency edge between the two variables if one exists, and null otherwise.

The user can modify the constraint system, and thus the dependency graph, in the following ways:

1. Create a new variable: This action adds a node to the dependency graph;
2. Delete a variable: This action deletes a variable that has no edges attached to it.
3. Add a new formula: This action attaches a formula to a variable. The formula does not have to declare its input variables, and therefore no new edges are created. The input variables are determined when the formula is actually evaluated, so dependency edges are added at evaluation time.
4. Delete a formula: This action removes a formula from a variable. All edges that enter the variable's node are destroyed, since the variable no longer requests the values of the variables that the edges originate from.
5. Change the value of a variable: This action changes the value of an variable. If the value of a constrained variable is changed, the constraint associated with that variable is temporarily unsatisfied. Allowing changes to constrained variables is useful for introducing new values into a cycle. For example, the constraints `a = b` and `b = a` lead to the cyclic dependency graph shown in Figure 7. Changing either `a` or `b` will cause the other variable to be changed as well, and allows us to simulate the multi-way constraint `a = b`.

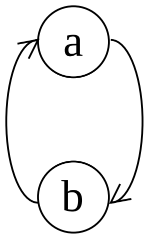


Figure 7: A circular dependency graph in which `a` has a formula that depends on `b` and `b` has a formula that depends on `a` (e.g., `a = b`, `b = a`). The editing model presented in this paper allows variables that contain formulas to be assigned temporary values. This temporary assignment allows new values to be introduced into cycles. For example, if `a` is given a new value, the value will be propagated automatically to `b`.

Multiple edits may be made before the constraint solver is called, and, adopting the terminology of Hudson [22], the set of edits between invocations of the constraint solver is called a *transaction*.

A variable's value is requested by calling `get_value` with the variable's name (`get_value` is considered an invocation of the constraint solver and thus ends the current transaction). `get_value` evaluates the constraint associated with a variable and then returns the variable's value. For example, the reference `self.prev.width` causes `get_value` to be called with `self.prev` as an argument (pointer variables are treated as first class citizens and thus their values can be computed using formulas). This returns a pointer to an object and `get_value` is called a second time to access the object's `width` instance variable[2].

Finally, to evaluate the time complexity of maintaining the dependency graph and finding the values of variables, we will use the following terms:

1. $directly\_affected_i$: The set of variables directly modified during transaction *i*.
2. $affected_i$: The set of variables that directly or indirectly depend on the changed variables in $\mathtt{directly\_affected}_i$.
3. $needed\_by\_affected_i$: The set of edges attached to the variables in $\mathtt{affected}_i$. $\mathtt{needed\_by\_affected}_i$ represents the set of dependencies needed to compute the values of the variables in $\mathtt{affected}_i$.
4. $added\_edges_i$: The set of dependency edges added during transaction *i*.
5. $no\_longer\_needed_i$: The set of dependency edges that become stale during transaction *i*. These dependency edges represent variables that the variables in $\mathtt{affected}_i$ depended on in previous transactions but which are no longer depended on in transaction *i*.

4.2 Lazy Evaluation

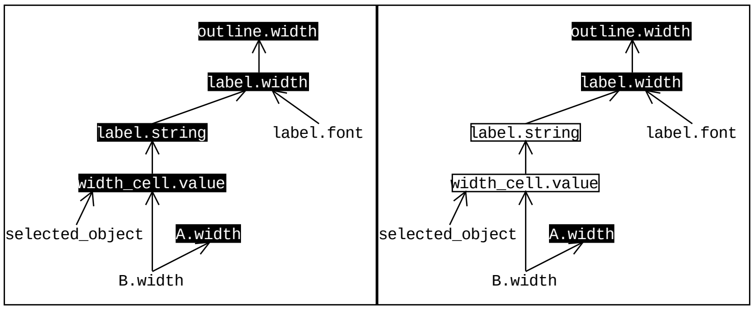
A lazy evaluation scheme for indirect reference constraints can be implemented using a variation of the nullification/reevaluation strategy [22, 40]. When the value of a variable changes, either by directly modifying the value or by installing a new formula in the variable, all variables that directly or indirectly depend on this changed variable are marked out-of-date (nullification phase). When the value of a variable is requested, the constraint that computes its value starts demanding the values of other variables on which it depends (reevaluation phase). If these variables are out-of-date, they will recursively demand the values of the variables they depend on, until eventually variables are reached whose values are up-to-date, at which point the constraints can compute their value and return [22, 40].

For example, suppose that the user changes the width of the selected object B in Figure 6.a. Figure 8.a shows the variables that are marked out-of-date as a result of this change. Note that all variables that can be reached from `B.width` are marked out-of-date. Figure 8.b shows the variables that will be reevaluated if the value of the label object's string attribute is requested. Note that only out-of-date variables whose values are needed to calculate the label's `string` are evaluated. Both the label's and outline rectangle's `width` are not required, and thus they remain out-of-date.

4.2.1 Maintaining the Dependency Graph

Nullification/reevaluation algorithms were originally constructed with the assumption that the edges in the graph remain static while the constraint solver is evaluating the graph. However, since the pointer variables may change, indirect reference constraints can cause the graph to dynamically change as the constraints are being evaluated, causing a constraint to access information from a different set of

[2] In practice, the reference `self.prev.width` would be compiled into the expression `get_value(get_value(self.prev).left)`.



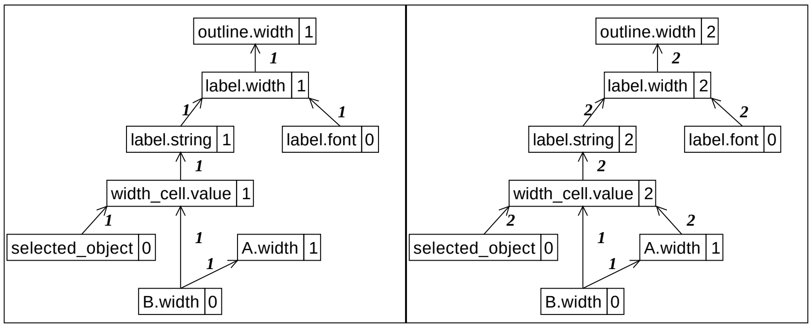
(a) (b)

Figure 8: (a) The black nodes denote the nodes marked out-of-date when the width of the selected object, `B.width`, is changed. Every node that can be reached from the changed node is marked. (b) The nodes surrounded by white rectangles represent the out-of-date variables in (a) that are reevaluated when the value of `label.string` is requested. Since `label.string` does not depend on either `label.width` or `outline.width`, the values of these variables are not demanded and thus remain out-of-date.

nodes. For example, when the constraint on node `width_cell.value` is evaluated after `selected_object` is changed from B to A, it starts referencing node `A.width` rather than node `B.width` (Figure 9.b).

To handle this situation, we have extended the algorithm so that dependencies can be dynamically created as the constraints are being evaluated and dynamically deleted as the constraints are being invalidated (alternative approaches are described in Section 6). The scheme is implemented by placing timestamps on each node and edge. The timestamp on a node represents the number of times the node has been evaluated. Each time the node is evaluated, the timestamp is incremented by one. The timestamp on an edge is the value of the timestamp on the node that the edge points to at the time the dependency was either created or last updated. A dependency's timestamp is updated whenever a node requests the value of the node that the dependency originates from. Figure 9.a shows the timestamps assigned to the dependency graph shown in Figure 6 after an initial evaluation. The timestamps are all 1's, since this is the first time each node has been evaluated. If the user changes the selection to object A, the dependency graph will be updated as shown in Figure 9.b. All the nodes that depended on `selected_object` are reevaluated, so their timestamps are incremented to 2. Similarly, the edges associated with demanded nodes have had their timestamps incremented to 2. Since the `selected_object` has changed from B to A, a new dependency edge has been added from `A.width` to `width_cell.value`.

To aid in dynamically creating such dependencies, the constraint solver uses a stack that keeps track of the variables whose formulas are currently being evaluated. Whenever a variable's formula starts executing, the variable is pushed onto the stack; thus the topmost variable will always be the variable whose formula is currently executing (the remaining formulas on the stack will have their execution temporarily suspended). When the constraint demands the value of a variable, it can create a dependency



(a) (b)

Figure 9: Two sample dataflow graphs with timestamps on the dependencies and the variables (the numbers in the boxes denote the variables' timestamps and the numbers on the edges denote the dependencies' timestamps). The dataflow graph in (a) represents the timestamps that would be applied to the dataflow graph in Figure 6.b. The dataflow graph in (b) represents the timestamps that are computed if `selected_object` is changed from B to A and all the variables are reevaluated. In dataflow graph (b), the dependency between `B.width` and `width_cell.value` is stale since the dependency's timestamp does not match `width_cell.value`'s time stamp.

by adding an edge from the demanded variable to the top variable on the stack. If there is no variable on the stack, then the user or an application is requesting the value and no dependency needs to be created.

In addition to dynamically creating new dependencies, the constraint solver must also dynamically remove stale dependencies. A dependency is considered stale if its timestamp does not match the timestamp on the node it points to. For example, `width_cell.value` no longer depends on `B.width` after the selected object is changed from B to A, so the timestamp on the edge between these nodes remains at 1 in Figure 9.b. The constraint solver removes stale dependencies as it invalidates constraints. Before following a dependency, it checks whether the dependency's timestamp matches the timestamp of the variable it points to. If the two timestamps disagree, the dependency is discarded. For example, when the value of `B.width` changes in Figure 9.b, the constraint solver will notice that the timestamp on the edge from `B.width` to `width_cell.value` does not match the timestamp on `width_cell.value`, and will remove the edge.

A beneficial side effect of this timestamping scheme is that constraints which involve conditionals depend only on the variables that make up the condition and the branch of the condition that is executed. Thus the number of dependency pointers and unnecessary evaluations are minimized. To see that this scheme works, note that a constraint will dynamically add or delete dependencies only if it contains pointers or conditionals. If a constraint depends on pointer variables, the constraint will be marked out-of-date when the pointer variables change and the constraint will be reevaluated when its value is next requested. At this point, the constraint solver will add edges to this constraint from the new set of nodes it references (Figure 9.b). The dependencies to variables that are not requested by the constraint on this evaluation will become stale and be removed the next time these dependencies are examined. Thus the constraint will demand the values of the correct set of nodes and will obtain the correct result.

In the case of a conditional, the branch or branches of the conditional that were ignored during the previous evaluation of the constraint will only have to be evaluated if the condition itself changes. For example, consider the following constraint:

```
d.left = if (launch_time > 0)
             then b.left + 10
             else c.left + 10
```

If the `launch_time` is greater than 0, then there will be dependency edges from `launch_time` and `b.left` to `d.left`, but not from `c.left` to `d.left`. If object `c` is moved, the constraint will not be reevaluated, which is correct, because `d.left` is not currently constrained to `c`.

Since the constraint depends on the variables in the condition, it will be marked out-of-date when one of these variables changes and will be automatically reevaluated (of course it will also be reevaluated if one of the variables in the branch that was last executed is changed). In the above example, the constraint will be marked out-of-date if either the launch time drops to 0 or below or object `b` moves. When a constraint is reevaluated, the constraint solver will add dependency edges to this constraint from the new set of variables it references in whichever branch is executed. The edges that emanate from variables in the previously executed branch will become stale and will be removed during a subsequent invalidation phase. Thus constraints with conditionals will always be evaluated correctly.

4.2.2 Data Structures

Two data structures are used to maintain the dependency graph—one for the variables (nodes) and one for the dependencies (edges). The variable data structure consists of five fields:

1. value: the current value of the variable;
2. outofdate: indicates if the value of the variable is out-of-date. Variables that are not computed by a formula are always considered up-to-date, and thus the outofdate field for these variables is always false.
3. dependencies: a list of dependencies that indicate which variables depend on this variable.
4. eval: a method (i.e., constraint) that computes the variable’s value. This field is null if a constraint is not attached to the variable.
5. timestamp: the number of times the eval method has been successfully executed.

When a dependency is created, it is stored in the `dependencies` list of the variable that is demanded by a constraint (i.e., the input variable) and it points to the variable that contains the constraint. The dependency data structure consists of two fields:

1. var: the variable that this edge is directed at.
2. timestamp: the value of `var`’s timestamp when this dependency was created or last updated.

4.2.3 Implementation

The algorithms that implement the nullification (`nullify`) and reevaluation (`get_value`) phases of the lazy evaluator are shown in Figures 10 and 11. A number of aspects of these algorithms merit special discussion. First, the manner in which the constraint system actually implements calls to `v.eval` in `get_value` is left unspecified because it is orthogonal to the implementation of pointer variables and thus existing algorithms for non-pointer variables can be used. The `eval` code can be as simple as a call to an expression written by the user. This is what is done in Garnet [33]. Alternatively, it could use Hudson's lazy evaluator [22]. In this case, `eval` would first request the values of the variables that the formula currently depends on and make sure that at least one of them has changed before evaluating the formula expression written by the user. The variable and dependency data structures would also have to be augmented with a number of additional fields to perform this test. See [22] for details on how to do this.

```
nullify( v : variable )
    v.outofdate = true
    for each dependency ∈ v.dependencies do
        if dependency.timestamp < dependency.var.timestamp then
            v.dependencies = v.dependencies - {dependency}
        else if dependency.var.outofdate = false then
            nullify( dependency.var )
```

Figure 10: `nullify` marks as out-of-date all variables that depend on a changed variable

```
get_value( v : variable )
    demanding_var = top(var_stack)
    if demanding_var then
        dependency = get_dependency(v,demanding_var)
        if dependency then
            dependency.timestamp = demanding_var.timestamp + 1
        else
            v.dependencies = v.dependencies ∪ <demanding_var,demanding_var.timestamp+1>
    if v.outofdate = true then
        push v onto var_stack
        v.outofdate = false
        v.value = v.eval()
        v.timestamp = v.timestamp + 1
        pop v off of var_stack
    return(v.value)
```

Figure 11: `get_value` performs two functions: 1) it sets up a new dependency or validates an existing dependency between two variables; and 2) it evaluates a variable's formula if it is out-of-date and returns the variable's value. If a variable does not have a formula, the `outofdate` flag is assumed to be always false. `var_stack` is used to keep track of variables that are being evaluated. The variable which is on top of the stack is the variable whose formula demanded `v`. `var_stack` is initially empty.

A variable's outofdate flag is set to true before a formula is evaluated to ensure that cycles terminate after one loop. If a variable is requested a second time, it will return its old value instead of trying to evaluate itself again, thus terminating the cycle.

In contrast, a variable's timestamp is updated *after* its formula has been evaluated. To ensure that dependencies are given a correct timestamp, a dependency's timestamp is set equal to the requesting variable's current timestamp, incremented by one. This deferred updating of variables' timestamps allows the constraint satisfier to gracefully recover if the formula crashes because of a user programming error. The ability to tolerate formula crashes is important in interpreted environments, such as development environments, debugging environments, and spreadsheet environments, where a user is allowed to fix an error and continue with a computation. If a variable's timestamp is updated before its formula is evaluated, and if the formula crashes before all the formula's dependencies have been updated, then the constraint satisfaction algorithm might mistakenly assume that the dependencies are stale and should be removed. For example, in Figure 9.a, suppose `label.string`'s timestamp is updated before `label.string`'s formula is evaluated, and further suppose that `label.string`'s formula crashes before the timestamp on the dependency between `width_cell.value` and `label.string` is updated. Then this dependency will be considered stale and will be eventually removed. In addition, `label.string`'s `outofdate` flag will be false, and thus the constraint solver will assume that `label.string` is up-to-date. Since the dependency will be removed, `label.string` will not be informed of subsequent changes to `width_cell.value`, and thus the width of the currently selected object will not be correctly displayed on the screen. Thus, to ensure that the system behaves properly even if formulas crash, it is necessary to update the timestamps after the node has been successfully evaluated.

4.2.4 Example

To illustrate how this algorithm works, consider what happens when the user changes the selection from object B to object A. This changes the value of `selected_object` and causes `nullify` to mark all variables reachable from `selected_object` as out-of-date (Figure 12.a). Since the timestamps on all edges visited by `nullify` are up-to-date, no edges are removed from the dependency graph. Next the screen manager updates the display, and requests the value of label's `string` variable (the screen manager needs to update several objects so its initial choice of variables to request is arbitrary). The sequence of calls to `get_value`, the value of the stack during each of these calls, the creation of edges in the dependency graph, and the updating of timestamps is shown in Figure 12. Since the value of `selected_object` has changed from B to A, a new edge is created from `A.width` to `width_cell.value`. In addition, since `width_cell.value` no longer requests the value of `B.width`, the timestamp on the edge between these two variables is not updated, and the dependency becomes stale. Eventually the display manager requests the values of the other out-of-date variables in the system, thus leading to the dependency graph shown in Figure 9.b.

4.2.5 Time Complexity

The time complexity of the lazy algorithm is determined by the number of variables marked out-of-date, the number of variables reevaluated, and the cost of maintaining the dependency graph. Maintaining the dependency graph includes updating the timestamps on dependencies and dynamically

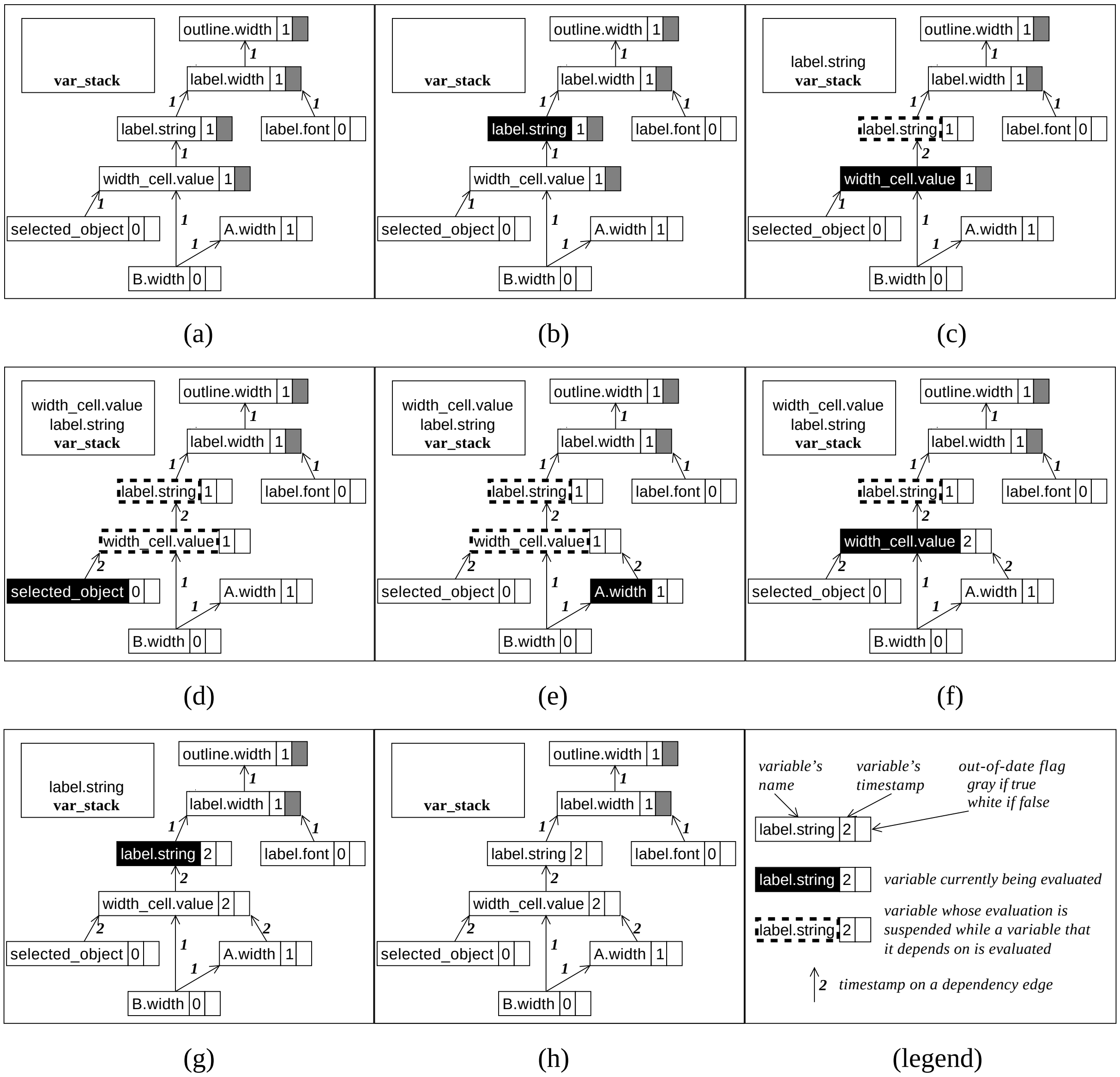


Figure 12: An example execution of the lazy evaluation algorithm. (a) The gray nodes denote the variables marked out-of-date when the `selected_object` is changed from B to A. (b) The value of `label.string` is requested, which causes the values of `width_cell.value`, `selected_object`, and `A.width` to be recursively requested (c-e). As variables with formulas are evaluated (`label.string` and `width_cell.value`), their `out_of_date` flags are marked false and they are pushed onto the `var_stack` that is used for creating dependencies. When variables are requested, the timestamps on existing dependency edges are updated (c-d) and new ones are created (e). As variables with formulas finish their evaluation, their timestamps are incremented, they are popped off the `var_stack`, and their values are returned (f-h).

adding and removing edges from the dependency graph. The time required to evaluate each equation is typically assumed to be O(1). Although this assumption may not always be true, it allows us to express the time complexity of the algorithm in terms of factors it can control, rather than factors beyond its control.

Because of lazy evaluation, not all of the variables in $\texttt{affected}_i$ will be evaluated during transaction *i* ($\texttt{affected}_i$ and the other terms used in this section are defined in Section 4.1). Similarly, the dependencies in $\texttt{no\_longer\_needed}_i$ will not become stale until the variables in $\texttt{affected}_i$ are evaluated, and will not be removed until later transactions. However, if we use an amortized analysis, then we can charge both the cost of evaluating a variable in $\texttt{affected}_i$ and the cost of removing dependencies that will eventually become stale to transaction *i*.

In the worst case, all the variables in $\texttt{affected}_i$ must be visited, marked out-of-date, and eventually brought up-to-date by evaluating them. The set of edges traversed during the invalidation phase is a subset of the edges in $\texttt{needed\_by\_affected}_i$, and thus the number of edges traversed is $O(|\texttt{needed\_by\_affected}_i|)$. The set of equations evaluated during the evaluation phase is $O(|\texttt{affected}_i|)$, since each equation in $\texttt{affected}_i$ is potentially evaluated.

Each of the dependencies in $\texttt{needed\_by\_affected}_i$ will either be updated or created during transaction *i*. To update or create a dependency edge, `get_value` needs to determine if an edge already exists. This involves searching through the dependencies associated with the demanded variable. In the worst case there will be an edge between the demanded variable and every other variable in the system. If there are *n* variables in the system, and the dependencies are arranged as a balanced tree, then each search will require $O(\texttt{log n})$ time. Thus the worst case time for updating and creating edges is $O(|\texttt{needed\_by\_affected}_i|\texttt{log n})$.

Finally, $|\texttt{no\_longer\_needed}_i|$ edges will become stale during transaction *i* and have to be removed. `nullify` can remove a stale edge in constant time, so the stale dependencies can be removed in $O(|\texttt{no\_longer\_needed}_i|)$ time.

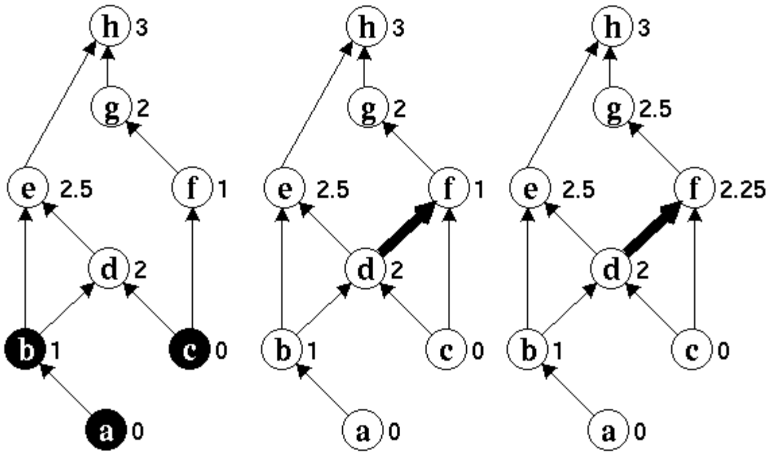
Summing up the various cost components of the lazy evaluator, we obtain a worst case running time of:

$$
\begin{aligned}
& O(|\texttt{needed\_by\_affected}_i| \\
& \quad + |\texttt{affected}_i| \\
& \quad + |\texttt{needed\_by\_affected}_i|\texttt{log n} \\
& \quad + |\texttt{no\_longer\_needed}_i|) \\
= \quad & O(|\texttt{affected}_i| + |\texttt{needed\_by\_affected}_i|\texttt{log n} + |\texttt{no\_longer\_needed}_i|)
\end{aligned}
$$

In actual applications, the number of edges that enter and leave a node is typically bounded by a relatively small constant, *k*. In this case, both $\texttt{needed\_by\_affected}_i$ and $\texttt{no\_longer\_needed}_i$ are bounded by $\texttt{k}|\texttt{affected}_i|$ and the `log n` term becomes a constant. Thus, the average time cost of the algorithm becomes $O(|\texttt{affected}_i|)$. Even this bound is unduly pessimistic because in most transactions, some of the variables in $\texttt{affected}_{\texttt{i}}$ will already be out-of-date and thus will not be charged to this transaction [22]. It is also important to note that for simplicity of presentation, we have not included the code or data structures that Hudson uses to prevent unnecessary evaluations of variables in [22]. However, it is not difficult to add this code or data structures to the algorithm presented in this paper, and thus evaluate the optimal number of variables.

4.3 Eager Evaluation without Cycles

The eager evaluation algorithm uses a variation of an eager evaluator developed by Roger Hoover [16]. Like the lazy evaluator, this algorithm makes use of dataflow graphs. However, it assigns position numbers to the nodes in the graph, indicating the nodes' relative position in topological order (Figure 13.a). A node's position number is always greater than the position numbers of any of its successors. When a node changes value, all of its immediate successors are added to a priority queue based on their position numbers. When the evaluator starts executing, it removes the variable with the lowest position number from the queue and evaluates it. By evaluating the lowest position numbered node in the queue, the evaluator ensures that the values of all the node's predecessors are up-to-date. For example, in Figure 13.a, the nodes `d`, `e`, and `f` are on the priority queue since they are successors of nodes `b` and `c`, which have been evaluated and which we assume have changed. Since `e` depends on `d`, `e`'s position number is greater than `d`'s, and thus `d` will be evaluated before `e` (`f` will be the first variable evaluated since it has the lowest position number of any variable on the evaluation queue).



(a) (b) (c)

Figure 13: (a) Numbers are assigned to nodes according to the order in which they are evaluated. Nodes cannot be evaluated until all their predecessors have been evaluated. Darkened nodes represent evaluated nodes. Nodes `d` and `f` are ready for evaluation. (b) Node `f` now depends on node `d` as well as node `c`. (c) Nodes `f` and `g` must be renumbered to make their position numbers agree with their position in topological order.

4.3.1 Maintaining the Dependency Graph

When an edge is added to a dataflow graph, the position numbers in the graph may have to be recomputed if the position numbers of the two nodes connected by the new edge are out-of-order. If the position numbers are out-of-order, the algorithm uses depth-first search to transitively follow the successors of the destination node until it reaches nodes whose position numbers are greater than the position number of the source node (data flows from the source node to the destination node; for example `e` is the source node and `h` is the destination node for the edge that connects these nodes). These nodes are termed boundary nodes. The algorithm works back from the boundary nodes and assigns new position numbers to the intermediate nodes that are between the position numbers of the source and boundary nodes.

For example, suppose a dependency from node `d` to `f` is added in Figure 13.b. Node `d` has a position number of 2 while node `f` has a position number of 1 so the nodes are out-of-order. The algorithm visits node `g`, which has a position number of 2, and then node `h`, which has a position number of 3. Since this position number is greater than node `d`'s position number, the search stops here and node `h` becomes a boundary node. In this case there is one existing position number, 2.5, between 2 and 3, so the algorithm assigns this position number to node `g`[3]. At this point the algorithm runs out of existing position numbers, so it creates a new one and assigns it to node `f` (Figure 13.c).

The Hoover algorithm assumes that dependency graphs cannot change once constraint evaluation begins, so the reordering scheme and the evaluator can be invoked in sequence. However, as explained in Section 4.2.1, indirect reference and conditional constraints may cause the edges of the graph to change *during* constraint evaluation. Thus the numbers assigned to the nodes may become incorrect and force an equation to be evaluated prematurely. To overcome this difficulty, we have taken the approach of dynamically updating the position numbers each time the graph changes, and evaluating nodes according to this revised topological order. In other words, the reordering algorithm and the constraint evaluator are interleaved.

New dependencies are created when a formula is evaluated. The same stack data structure used to create dependencies in lazy evaluation is used to create dependencies in eager evaluation. When creating a new dependency, it is necessary to check that the position number of the variable whose formula is requesting the value is greater than the position number of the variable whose value is being requested. This is done to ensure that the variables are in the correct topological order. If the requested variable's position number is greater than or equal to the requesting variable's position number, then the position numbers are recomputed as described above.

Once the reordering is complete, the evaluator must check whether the position number of the variable that is being evaluated (i.e., the requesting variable) has become greater than the smallest position number on the evaluation queue. If it has, the variable's evaluation must be stopped since it is potentially being evaluated out-of-order. The variable will be popped off the dependency stack, reinserted

[3]We are using rational numbers for illustrative purposes only. The actual algorithm for creating position numbers uses integers. See Section 4.3.2 for more details about the algorithm.

into the evaluation queue, and its evaluation aborted (in this algorithm, nested evaluations cannot happen if there are no cycles, so this formula is the only formula whose evaluation will have to be aborted). Since the evaluation of constraints is assumed not to have side-effects, aborting the evaluation should not have any adverse consequences[4]. When the variable is evaluated again, its formula will be evaluated from the beginning.

Dependencies are removed at two different points during constraint evaluation. First, when the eager evaluator adds a variable's successors to the priority queue, it removes any edges whose timestamps do not match the timestamp of the node they point to. Second, the eager evaluator discards any out-of-date dependency edges that it encounters when it recomputes position numbers.

4.3.2 Data Structures

As in lazy evaluation, two data structures are used to maintain the dependency graph—one for the variables (nodes) and one for the dependencies (edges). The data structure for dependencies is identical to the one presented in Section 4.2.2. The data structure for variables is similar to the one presented in Section 4.2.2, but has two differences: 1) a `position_number` field is added indicating a variable's position in topological order and; 2) the `outofdate` field is omitted, since all potentially out-of-date variables are brought up-to-date during eager evaluation.

It might seem necessary to add a `visited` field to the variable's data structure, since recomputing the position numbers requires a depth-first search. However, once all of a node's successors have been visited, it is possible to assign a new position number to the node. This new position number will be greater than the position number of the original demanded variable, and thus the node will become a boundary node. As a result, the depth-first search will stop at this node on any future visits. In effect, the new position number is a visited mark, and thus an explicit `visited` field is not necessary.

The position numbers are maintained in an ordered list using an algorithm described in [45]. Each node in the dataflow graph points to one of these position numbers. Comparisons between position numbers can be performed in O(1) time and insertions of new position numbers can be accomplished in amortized O(1) time, provided that numbers of `log n` bits can be manipulated in O(1) time by the underlying hardware. In this paper, the function `position_number_between` is used to generate these position numbers. It takes two position numbers, `x` and `y`, `x < y`, as arguments, and returns a position number between `x` and `y` as the result.

4.3.3 Implementation

The eager evaluator without cycles is shown in Figures 14-16. Whenever a formula is stored in a variable, the variable is placed on the evaluation queue (`eval_q`). If a regular value is stored in the variable instead, the variables that depend on the changed variable are placed on the evaluation queue.

When users are ready to have the solution to the constraint system updated, they call

[4] Actually side-effects are permissible if all input variables have been read and it is assured that the formula will not be aborted. This allows constraints such as the one shown in Section 2.2 to be supported.

propagate. propagate removes variables from the evaluation queue in topological order and evaluates them. It also maintains the dependency stack and priority queue, and updates timestamps.

get_value is responsible for obtaining the value of a variable, and for adding or updating dependencies. The portion of get_value that returns the value of the variable is significantly shorter than the lazy version of get_value, since the requested variable's value is always up-to-date and thus can be returned immediately. This is guaranteed by the evaluation of variables in topological order.

reorder is responsible for recomputing the position numbers when get_value creates a new dependency and finds that the two variables involved in the dependency are out-of-order. reorder takes two parameters, the position number of the variable that was originally requested (this is the variable that the dependency emanates from), and the variable currently being visited in the depth-first search. A new position number is assigned to the variable that is between the position number passed in and the minimum of the variable's successors' position numbers.

```
propagate(eval_q : priority-queue)
    while eval_q ≠ null do
        v = delete_min(eval_q)
        push v onto var_stack
        temp = v.value
        v.value = v.eval()
        if v.value ≠ temp then
            /* get rid of stale dependencies */
            v.dependencies = v.dependencies -
                    {w | w ∈ v.dependencies and w.timestamp < w.var.timestamp}
            eval_q = eval_q ∪ { w.var | w ∈ v.dependencies }
        v.timestamp = v.timestamp + 1
```

Figure 14: propagate is called when the user wants the solution to the constraint system updated. eval_q initially consists of the set of variables that have been assigned new constraints, or which depend on a variable that has been assigned a new value.

4.3.4 Example

To illustrate the eager evaluation algorithm, assume that the user changes the selection from object B to object A in Figure 6. Figure 17 traces the sequence of calls to get_value, the updates to the timestamps, and the values of the priority queue (eval_q) and dependency stack (var_stack) as the eager evaluator executes. Changing the selected object causes width_cell.value to be placed on the evaluation queue, since it is a successor of the changed variable, selected_object. Constraint satisfaction begins with width_cell.value since it is the only variable on the evaluation queue. As width_cell.value is evaluated, it demands the values of selected_object and A.width. The dependency between selected_object and width_cell.value already exists, but a new dependency must be created between A.width and width_cell.value. Since this new dependency connects two variables whose position numbers are out-of-order, reorder is called to update the position numbers. When reorder finishes, there are no variables in the evaluation queue, and thus the evaluation of width_cell.value continues (if there were variables in the queue with lower position

```
get_value( v : variable )
    demanding_var = top(var_stack)
    if demanding_var then
        dependency = get_dependency(v,demanding_var)
        if dependency then
            dependency.timestamp = demanding_var.timestamp + 1
        else
            v.dependencies = v.dependencies ∪ <demanding_var,demanding_var.timestamp+1>
            if v.position_number ≥ demanding_var.position_number
                reorder(v.position_number,demanding_var)
                if demanding_var.position_number > min(eval_q).position_number then
                    eval_q = eval_q ∪ {demanding_var}
                    pop demanding_var off of var_stack
                    abort demanding_var.eval

    return(v.value)
```

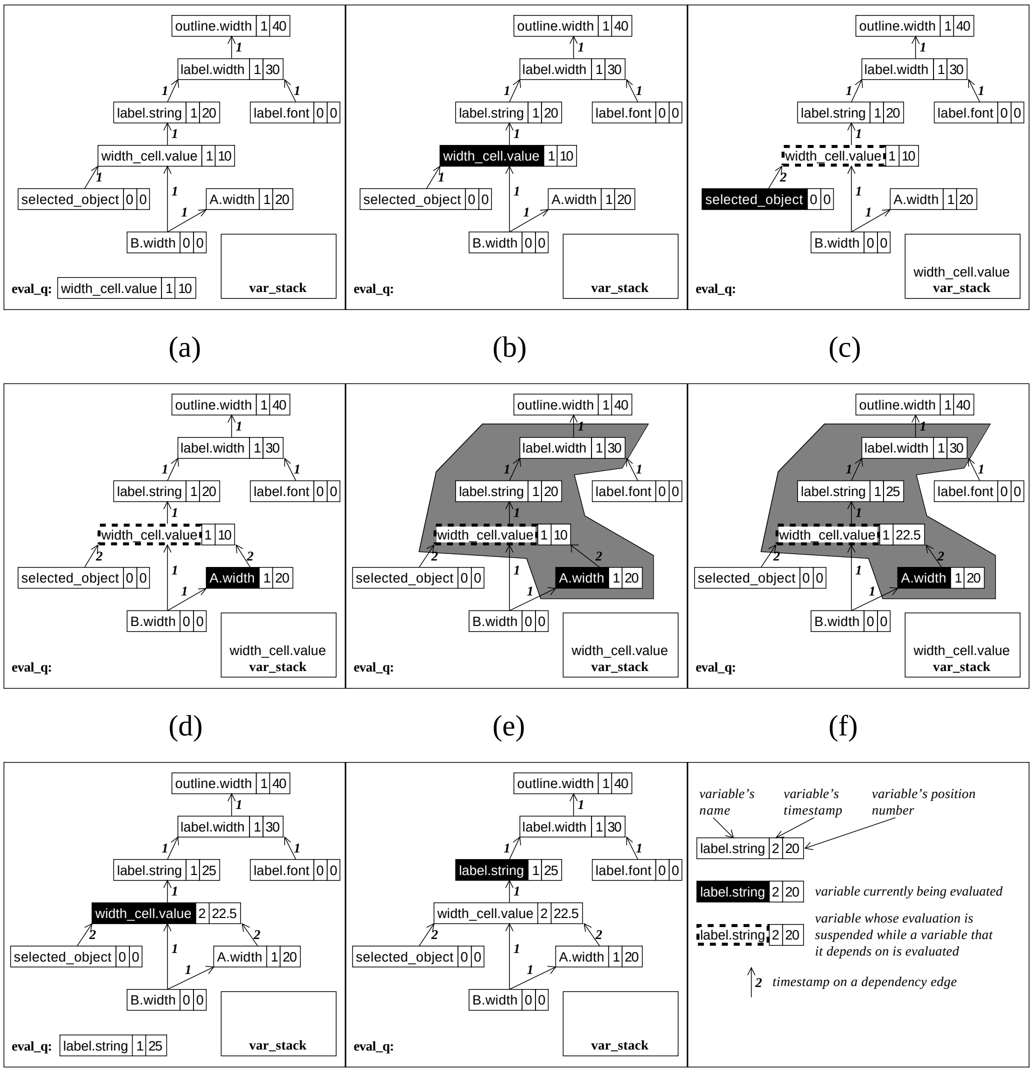
Figure 15: `get_value` performs two functions: 1) it sets up a new dependency or validates an existing dependency between two variables; and 2) it returns the variable's value. When a new dependency is established, `get_value` ensures that the position numbers of the variables are in order, and if they are not, calls `reorder` to update the position numbers. If this reordering causes the position number of the currently executing variable to exceed the minimum position number on the evaluation queue, then execution of the variable is terminated.

```
reorder( position_number : integer; v : variable )

    ceiling = *max_position_number*   /* largest possible position number */
    for each dependency ∈ v.dependencies do
        w = dependency.var
        if dependency.timestamp < w.timestamp then
            w.dependencies = w.dependencies - {dependency}
        else if w.position_number ≤ position_number then
            ceiling = min(ceiling, reorder(position_number, w))
        else
            ceiling = min(ceiling, w.position_number)
    v.position_number = position_number_between(position_number, ceiling)
    return(v.position_number)
```

Figure 16: `reorder` assigns new position numbers to variables by finding the minimum position number of a variable's successors and assigning a position number between the position number passed to `reorder` and this minimum position number.

numbers, the evaluation of `width_cell.value` would be aborted and `width_cell.value` would be reinserted into the evaluation queue). The remaining portion of the constraint satisfaction process proceeds without any new dependencies being created, and it quiesces when the evaluation of `label.width` is completed. The figure assumes that `label.width's` value does not change and therefore it is not necessary to evaluate or examine `outside.width`.

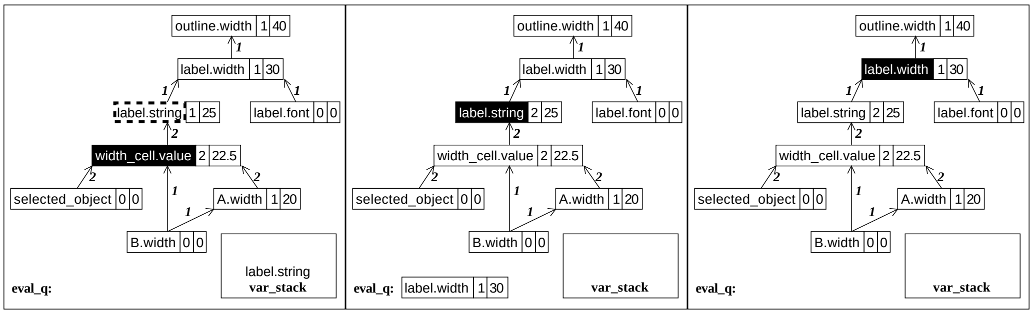


(a) (b) (c)

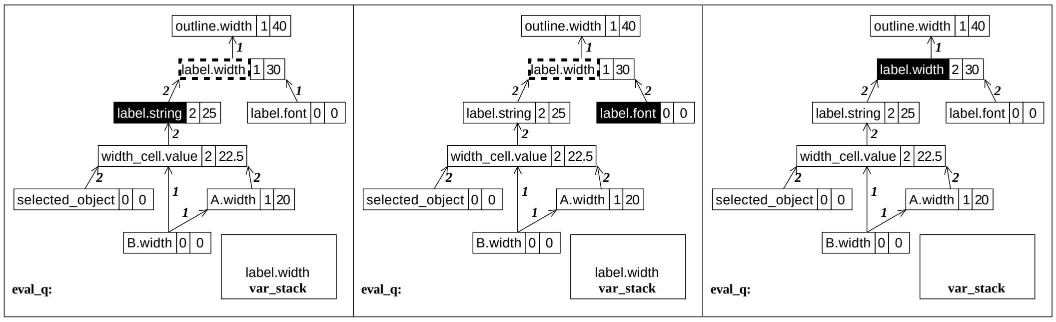
(d) (e) (f)

(g) (h) (legend)

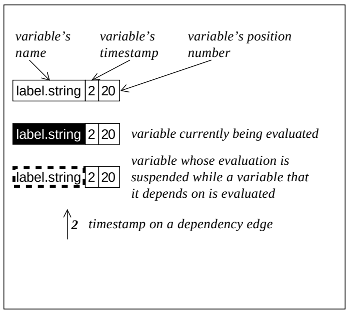
Figure 17: (Continued on next page). An example execution of the eager evaluator. (a) `selected_object` has been changed so that it points to object A. This causes `width_cell.value` to be placed on the evaluation queue. (b) The eager evaluator starts evaluating `width_cell.value`, which requests the values of `selected_obj` (c) and `A.width` (d). A new dependency is created between `A.width` and `width_cell.value`, which causes the two variables' position numbers to become out-of-order. `reorder` is called to recompute the position numbers and visits the nodes in the gray polygon (e). The position numbers of the nodes in this polygon are updated, the evaluation of `width_cell.value` is aborted, and `width_cell.value` is placed back on the evaluation queue (f). Since `width_cell.value` is the only variable on the priority queue, it is again evaluated and this time its evaluation terminates normally (g). `label.string` and `label.width` are then evaluated and the algorithm terminates (h-n). As variables are evaluated they are pushed onto the variable stack so that dependencies can be created. When variables are requested, dependencies are either updated (e.g., (c)) or created (e.g., (d)). When the evaluation of variables is completed, they are popped off the variable stack, their timestamps are incremented, and their values are returned.



(i) (j) (k)



(l) (m) (n)



(legend)

Figure 17 continued.

4.3.5 Time Complexity

The time complexity of the eager evaluation algorithm is determined by 1) the cost of maintaining the dependency graph, 2) the cost of updating the position numbers, and 3) the number of equations that must be reevaluated. We will again assume that the time required to evaluate an equation is O(1).

The cost of maintaining the dependency graph includes the cost of creating and updating dependency edges as equations are evaluated, and removing stale dependency edges that are created during the evaluation. As with the analysis for lazy evaluation, we will charge the cost of removing stale dependency edges against transaction *i*, even though the edges will be removed during later transactions.

Each of the dependency edges in $\mathtt{needed\_by\_affected}_\mathtt{i}$ will be either updated or created during transaction *i* ($\mathtt{needed\_by\_affected}_\mathtt{i}$ and the other terms used in this section are defined in Section 4.1). Each created edge might cause an equation evaluation to be aborted and the equation to be reinserted in the evaluation queue. Aborted equations will have to be reevaluated, which can cause dependencies to be redundantly updated. For each reevaluated equation, the set of updated dependencies is a subset of $\mathtt{needed\_by\_affected}_\mathtt{i}$, and thus each added edge can result in an additional $O(|\mathtt{needed\_by\_affected}_\mathtt{i}|)$ dependency edges being updated. In order to create or update a dependency, it is first necessary to determine whether a dependency already exists. This requires searching through the dependencies of the demanded variable and may require up to $O(\mathtt{log\ n})$ time if the demanded variable has a directed edge to every other variable in the dependency graph. The total cost of updating and creating dependencies is therefore $O((|\mathtt{needed\_by\_affected}_\mathtt{i}| + |\mathtt{added\_edges}_\mathtt{i}|\, |\mathtt{needed\_by\_affected}_\mathtt{i}|)\ \mathtt{log\ n})$, which can be simplified to $O(|\mathtt{added\_edges}_\mathtt{i}|\, |\mathtt{needed\_by\_affected}_\mathtt{i}|\ \mathtt{log\ n})$.

The number of stale dependencies created is $|\mathtt{no\_longer\_needed}_\mathtt{i}|$. As in the lazy algorithm, these dependencies can be removed in constant time and thus the cost of removing them is $O(|\mathtt{no\_longer\_needed}_\mathtt{i}|)$.

The worst case for updating position numbers occurs when each added edge connects two variables whose position numbers are out-of-order and causes each variable in $\mathtt{affected}_\mathtt{i}$ to have a new position number assigned to it. The set of edges that the depth-first search will traverse while updating the position numbers is a subset of $\mathtt{needed\_by\_affected}_\mathtt{i}$. Thus, each added edge may require $O(|\mathtt{needed\_by\_affected}_\mathtt{i}|)$ time to update the position numbers, since the depth-first search used by this update process is linear in the number of edges traversed. The total time to update the position numbers is therefore $O(|\mathtt{added\_edges}_\mathtt{i}|\, |\mathtt{needed\_by\_affected}_\mathtt{i}|)$.

Each added edge could also cause an equation evaluation to be aborted and the equation to be placed back on the evaluation queue. Thus up to $(|\mathtt{added\_edges}_\mathtt{i}| + |\mathtt{affected}_\mathtt{i}|)$ equations may have to be reevaluated. The equations are maintained in a priority queue, and there may be up to $(|\mathtt{added\_edges}_\mathtt{i}| + |\mathtt{affected}_\mathtt{i}|)$ insertions and $(|\mathtt{added\_edges}_\mathtt{i}| + |\mathtt{affected}_\mathtt{i}|)$ deletions to the priority queue. Since each insertion or deletion requires $O(\mathtt{log}\ |\mathtt{affected}_\mathtt{i}|)$ time, the time required to reevaluate equations is $O((|\mathtt{added\_edges}_\mathtt{i}| + |\mathtt{affected}_\mathtt{i}|)\mathtt{log}\ |\mathtt{affected}_\mathtt{i}|)$.

The cost of creating and updating dependencies dominates the cost of reevaluating equations, so the worst case running time of the eager evaluator is determined by summing up the cost of maintaining the dependency graph and the position numbers. This gives us a bound of

$$
\begin{aligned}
& O(|\mathtt{added\_edges}_\mathtt{i}|\, |\mathtt{needed\_by\_affected}_\mathtt{i}|\ \mathtt{log\ n} \\
& \quad + |\mathtt{no\_longer\_needed}_\mathtt{i}| \\
& \quad + |\mathtt{added\_edges}_\mathtt{i}|\, |\mathtt{needed\_by\_affected}_\mathtt{i}|) \\
= \quad & O(|\mathtt{added\_edges}_\mathtt{i}|\, |\mathtt{needed\_by\_affected}_\mathtt{i}|\ \mathtt{log\ n} \\
& \quad + |\mathtt{no\_longer\_needed}_\mathtt{i}|)
\end{aligned}
$$

This worst case bound is a very pessimistic estimate of the running time of the algorithm. As noted in Section 4.2.5, the number of edges that enter and leave a node is typically bounded by a relatively small constant in actual applications. Thus the number of edges in $\mathtt{needed\_by\_affected}_i$ and $\mathtt{no\_longer\_needed}_i$ are proportional to $|\mathtt{affected}_i|$, and the `log n` factor becomes a constant. In addition, the number of variables in the evaluation queue at any one time and the number of edges added tend to be bounded by a relatively small constant. In this case, the average running times of all the cost components decreases to $O(|\mathtt{affected}_i|)$, which becomes the average case running time of the algorithm as well.

Even this bound is unduly pessimistic because in most transactions, not all of the variables will have to be reevaluated or assigned new position numbers. Indeed, results based on testing of the algorithm suggest that pointer variables typically do one of two things: 1) they shift between nodes whose position numbers are identical, thus causing no reordering to occur; or 2) they shift between a fixed set of nodes, and once they have shifted to the highest numbered node, reordering never occurs again. The former case arises frequently in animations where an object is moving between independent but fairly similar objects that have roughly the same number of constraints. The latter case arises frequently in menus where the last item has the constraints with the highest position number (because it is the last item laid out). Thus in practice, the algorithm appears to fairly rapidly quiesce to a state where very few reorderings occur during constraint evaluation.

4.4 Eager Evaluation with Cycles

The algorithm presented in the previous section cannot handle cycles because `reorder` generates correct position numbers only if the graph is acyclic. However, the algorithm can be modified to handle cycles. Conceptually, this modification can be made by collapsing cycles into a single node, with each of the equations in the cycle having the same position number (Figure 18). Variables that are not in a cycle are evaluated as they were when cycles were not allowed. Variables in a cycle are evaluated using the nullification/reevaluation algorithm described in Section 4.2 (the rationale for using nullification/reevaluation to evaluate cycle nodes is discussed in Section 4.4.6). Thus, when the evaluator encounters a variable that belongs to a cycle, it marks all variables in the cycle as out-of-date, and then demands their values. This results in each variable in the cycle being evaluated once. If the constraints are inconsistent (e.g., `A = B + 10`, `B = A + 10`), then one or more constraints may be left unsatisfied.

4.4.1 Maintaining the Dependency Graph

Each new dependency that is created when a formula is evaluated has the potential to create a new cycle, and each dependency that is deleted has the potential to break a cycle. Cycles are detected using an algorithm that finds the strongly connected components of a graph [1, pp. 189-195]. A strongly connected component is a set of nodes such that for every pair of nodes `v` and `w`, there is a path from `v` to `w` and a path from `w` to `v`. Any strongly connected component of size greater than one is a cycle.

When a cycle is created, the same position number is assigned to each node in the cycle. The position number is computed by finding the minimum position number among all successors of nodes in the cycle, and then assigning all the nodes in the cycle a position number that is less than this minimum position number. For example, in Figure 18, the cycle consisting of `b`, `c`, `d`, and `e` has only one successor, `f`, which has a position number of 3. Thus the variables in this cycle are all assigned a position number less than 3, in this case 2. When a cycle is broken, the position numbers of the nodes in the former cycle must be updated so that they accurately reflect their new position in topological order. For example, if the edge from `i` to `g` is deleted in Figure 18, then a new position number of 5 might be assigned to `h` and a new position number of 6 might be assigned to `i`.

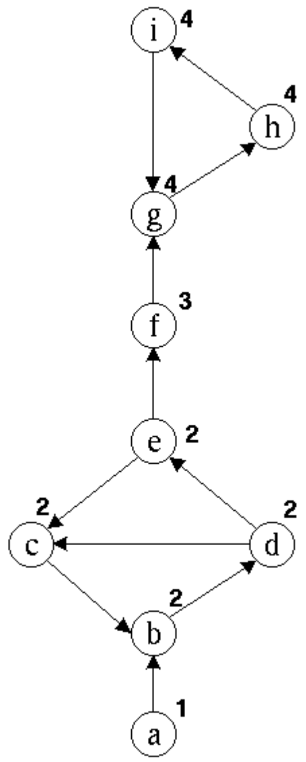


Figure 18: Conceptually, the eager evaluator collapses nodes in a cycle into a single node. It does this by assigning the same position number to each variable in a cycle.

As in eager evaluation without cycles, edges are created when a formula is evaluated, and edges are lazily removed when position numbers are updated or when variables are added to the evaluation queue. Edges are also removed just before a cycle is evaluated. This removal is necessary since some of the dependencies in the cycle may be stale and thus the cycle may no longer exist. If this is the case, then normal evaluation should be used instead of cycle evaluation. For example, in Figure 19.a, the dependency between nodes `a` and `c` is stale. Thus it should be removed, the position numbers should be updated, and the variables should be evaluated using normal evaluation.

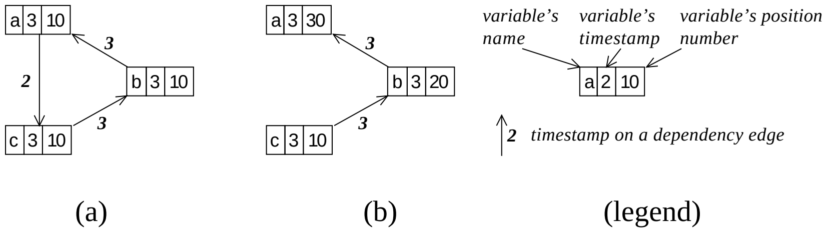


Figure 19: Stale dependencies are lazily removed, so it is important to check that a variable is still part of a cycle before performing cycle evaluation. In (a) there is a stale dependency between `a` and `c` (shown by the disparity in timestamps) and thus the variables no longer belong to a cycle. If `reorder` is called before cycle evaluation is done, it will detect that the variables are no longer part of a cycle and assign them new position numbers. This will allow normal evaluation to occur. (b) shows the updated dataflow graph.

4.4.2 Data Structures

The eager evaluator that handles cycles uses the same data structure for dependencies that it used when cycles were not permitted and extends the data structure it used for variables. Three new fields, `visited`, `dfnumber`, and `lowlink` are added to the variable data structure to handle cycle detection. The use of these fields is described in the next section. The `outofdate` field that was used by the lazy evaluator is added as well, since cycles are evaluated using nullification/reevaluation.

4.4.3 Implementation

The eager evaluator with cycles is shown in Figures 20-22. Each of the three main procedures, `propagate`, `get_value`, and `reorder`, has been somewhat modified. `propagate` has been modified so that it evaluates cyclic variables using a nullification/reevaluation strategy. If it detects that a variable in a cycle is about to be evaluated, it calls `reorder` to determine whether the cycle still exists. As discussed in Section 4.4.1, this check is necessary since dependencies are lazily removed, and thus the fact that the cycle has been broken may not yet have been detected. If the cycle has been broken, the position numbers of the variables in the cycle will be recomputed, and this renumbering might make the position number of the variable greater than the position number of some variable on the evaluation queue, thus making the evaluation of the variable premature. Thus line 5 of `propagate` checks whether the variable should be evaluated or reinserted into the evaluation queue. If it is appropriate to evaluate the variable, `propagate` uses nullification/reevaluation if the variable is still part of a cycle, and normal evaluation otherwise.

`get_value` has been modified so that instead of automatically returning the value of a variable, it first checks whether the variable has been marked out-of-date. If the variable is marked out-of-date, then `get_value` uses a nullification/reevaluation strategy to reevaluate the variable. The code here is the same code used to evaluate variables in the lazy evaluator (Figure 11). The portion of `get_value` that creates and updates dependencies has not been changed.

```
propagate( eval_q : priority-queue)
    while eval_q ≠ null do
        v = delete-min(eval_q)
        if v.cycle = true then
            reorder_variables(v,v)
            if v.position_number ≥ min(eval_q).position_number then
                insert(v,eval_q)
            else if v.outofdate = false then
                normal_eval(v)
            else
                get_value(v)
        else
            normal_eval(v)

normal_eval( v : variable )
    push v onto var_stack
    temp = v.value
    v.value = v.eval()
    if v.value ≠ temp then
    /* get rid of stale dependencies */
        v.dependencies = v.dependencies -
                                  {w | w ∈ v.dependencies and w.timestamp < w.var.timestamp}
        eval_q = eval_q ∪ { w.var | w ∈ v.dependencies }
    v.timestamp = v.timestamp + 1
```

Figure 20: `propagate` handles cyclic variables by first ensuring that they are still part of a cycle, and, if they are, then calling `get_value` which uses a nullification/reevaluation scheme to evaluate them. The handling of non-cyclic variables is not altered.

`reorder` has been modified so that its depth-first search is capable of detecting and handling cycles. The backbone of the algorithm is shown in Figure 23 (the algorithm detects strongly connected components and is adapted from [1, pp. 189-195]), while the complete algorithm that detects cycles and assigns position numbers is shown in Figure 22. In both figures, a *dfnumber* of *i* indicates that this is the i*th* node visited during the depth-first search. *lowlink* is defined as ([1], 190):

```
v.lowlink = min({v.dfnumber}
            ∪ {w.dfnumber | there is a cross edge or back edge from a
                  descendant of v to w, and the root of the strongly
                  connected component containing w is an ancestor of v}
```

The definition of lowlink implies that if a variable's lowlink *j* is lower than its dfnumber *i*, then the variable belongs to a strongly connected component involving *j*, and thus belongs to a cycle. A variable may also belong to a cycle if its lowlink equals its dfnumber. In this case, the variable would be the ‘‘root’’ of the cycle (see for example, variables `b` and `g` in Figure 24.a). The dfnumbers and lowlinks that this algorithm might compute for the graph in Figure 18 are shown in Figure 24.a.

`reorder` modifies the algorithm shown in Figure 23 in a number of ways. First, it is only necessary to search to the boundary nodes of the graph rather than the entire graph when an edge is added.

```
get_value( v : variable )
    demanding_var = top(var_stack)
    if demanding_var and demanding_var ≠ v then
        dependency = get_dependency(v,demanding_var)
        if dependency then
            dependency.timestamp = demanding_var.timestamp + 1
        else
            v.dependencies = v.dependencies
                              ∪ <demanding_var,demanding_var.timestamp+1>
            if v.position_number ≥ demanding_var.position_number then
                reorder_variables(v,demanding_var)
                if demanding_var.position_number > min(eval_q).position_number or
                    v.position_number = demanding_var.position_number then
                    eval_q = eval_q ∪ {demanding_var}
                    pop demanding_var off of var_stack
                    abort demanding_var.eval
    if v.outofdate = true then
        push v onto var_stack
        v.outofdate = false
        temp = v.value
        v.value = v.eval()
        pop v off of var_stack
        if v.value ≠ temp then
            /* get rid of stale dependencies */
            v.dependencies = v.dependencies -
                    {w | w ∈ v.dependencies and w.timestamp < w.var.timestamp}
            eval_q = eval_q ∪ { w.var | w ∈ v.dependencies
                              and (w.var.position_number ≠ v.position_number
                                   or w.var.outofdate = true) }
        v.timestamp = v.timestamp + 1
    return(v.value)
```

Figure 21: `get_value` evaluates a variable's formula if it is out-of-date and returns the variable's value. A variable's `outofdate` flag will be true only if the variable belongs to a cycle that is being evaluated. If a variable is self-circular (i.e., it demands itself, thus failing the test `demanding_var ≠ v`), its old value is simply returned and it is not marked as being part of a cycle. It is more efficient to treat a self-circular variable as being non-circular, and handling it by returning its old value produces the same result as evaluating it using a nullification/reevaluation strategy. The condition `w.position_number ≠ v.position_number or w.outofdate = true` prevents variables that are part of the same cycle from being added to the evaluation queue. If the cycle has been broken, then the variable's dependents may have the same position number but still be marked out-of-date. In this case the dependents are part of the old cycle and should be added to the evaluation queue. The abort command terminates execution of `demanding_var`'s formula and any nested evaluations. All formulas whose evaluation is aborted will be started from the beginning when they are reevaluated.

Recall that a boundary node is a node whose position number is greater than the position number of the node from which the new edge emanates. Thus the conditional on lines 8-12 of the strong-connectivity algorithm can be encapsulated in another conditional which tests whether the current node is a boundary

```
reorder_variables( initial_requested_var : variable; v : variable )
1       count = 1
2       reorder_stack = null
3       reorder(initial_requested_var.position_number,v)

reorder( position_number : integer; v : variable )
1       v.visited = true
2       v.dfnumber = count
3       count = count + 1
4       v.lowlink = v.dfnumber
5       ceiling = *max_position_number*  /* largest possible position number */
6       push v onto reorder_stack
7       for each dependency ∈ v.dependencies do
8               w = dependency.var
9               if dependency.timestamp < w.timestamp then
10                      v.dependencies = v.dependencies - {dependency}
11              else if w.position_number ≤ position_number then
12                      if w.visited = false then
13                              ceiling = min(ceiling, reorder(position_number, w))
14                              v.lowlink = min(v.lowlink, w.lowlink)
15                      else
16                              v.lowlink = min(v.lowlink, w.dfnumber)
17              else
18                      ceiling = min(ceiling, w.position_number)
19      if v.lowlink = v.dfnumber then
20              cycle_flag = if top(reorder_stack) ≠ v then true else false
21              initial_cycle_flag = if cycle_flag and (v.dfnumber = 1) then true else false
22              new_position_number = if initial_cycle_flag and initial_requested_var.visited = true
23                       then position_number
24                       else position_number_between(position_number, ceiling)
25              outofdate_flag = if initial_cycle_flag and
26                                      (new_position_number ≤ min(eval_q).position_number)
27                              then true else false
28              repeat
29                      x = pop(reorder_stack)
30                      x.cycle = cycle_flag
31                      x.position_number = new_position_number
32                      x.visited = false
33                      x.outofdate = outofdate_flag
34              until x = v
35              ceiling = new_position_number
36      return(ceiling)
```

Figure 22: `reorder` uses a strong connectivity algorithm to detect cycles in the constraint graph and to assign position numbers to variables that reflect the variable's position in topological order. If `reorder` detects a cycle that is about to be evaluated, it also sets the `outofdate` flags in the variables that comprise the cycle.

node (lines 11-18 of `reorder`). Note that when cycles were not permitted, the boundary node test allowed us to eliminate the `visited` test because a node could not be revisited until the depth-first search starting at the node had been completed. When the depth-first search completed, a new position

```
strong-connectivity( v : variable )
1       v.visited = true
2       v.dfnumber = count
3       count = count + 1
4       v.lowlink = v.dfnumber
5       push v onto stack
6       for each dependency ∈ v.dependencies do
7               w = dependency.var
8               if w.visited = false then
9                       strong-connectivity(w)
10                      v.lowlink = min(v.lowlink, w.lowlink)
11              else if w.dfnumber < v.dfnumber and w is on the stack then
12                      v.lowlink = min(v.lowlink, w.dfnumber)
13      if v.lowlink = v.dfnumber then
14              print "strongly-connected component = {"
15              repeat
16                      x = pop(stack)
17                      print x
18              until x = v
19              print "}"
```

Figure 23: The above algorithm enumerates the strongly connected components of a graph. `reorder` uses a modified version of this algorithm to detect cycles and assign position numbers to variables. `count` and `stack` are global variables which are initialized to 1 and empty respectively before `strong-connectivity` is called.

number was assigned to the node, making it a boundary node. Once a node was a boundary node, it could not be revisited, and thus the `visited` test was superfluous. However, when cycles are allowed, a node that is part of a cycle can be revisited before the depth-first search that starts at that node is completed. To avoid revisiting this node, the `visited` test must be retained.

As in eager evaluation without cycles, `reorder` also determines the minimum position number of a node's or cycle's successors during the depth-first search. In Section 4.4, a node looked at each of its successors to determine its minimum position number. However, a node in a cycle should only examine those successors which do not belong to the same cycle. These successors will be boundary nodes, since they will already have had new position numbers assigned to them. Once a cycle node has computed the minimum position number of its successors, it should propagate this information back to the root node of the cycle, so that a common position number for all the nodes in the cycle can be computed. The root node of the cycle is the first node in the cycle that the depth-first search visited. The minimum position number of a node's successors can be propagated back toward the root cycle node when the depth-first search of the node finishes. Thus `reorder` replaces the recursive depth-first search call on line 9 of `strong-connectivity`, with a statement that first calls depth-first search and then computes the lesser of the position number returned by this call and the current minimum position number (line 13 of `reorder`).

If the node is revisited while the depth-first search of the node is still in progress, then the minimum position number information is incomplete, and thus should not be returned. Thus no position

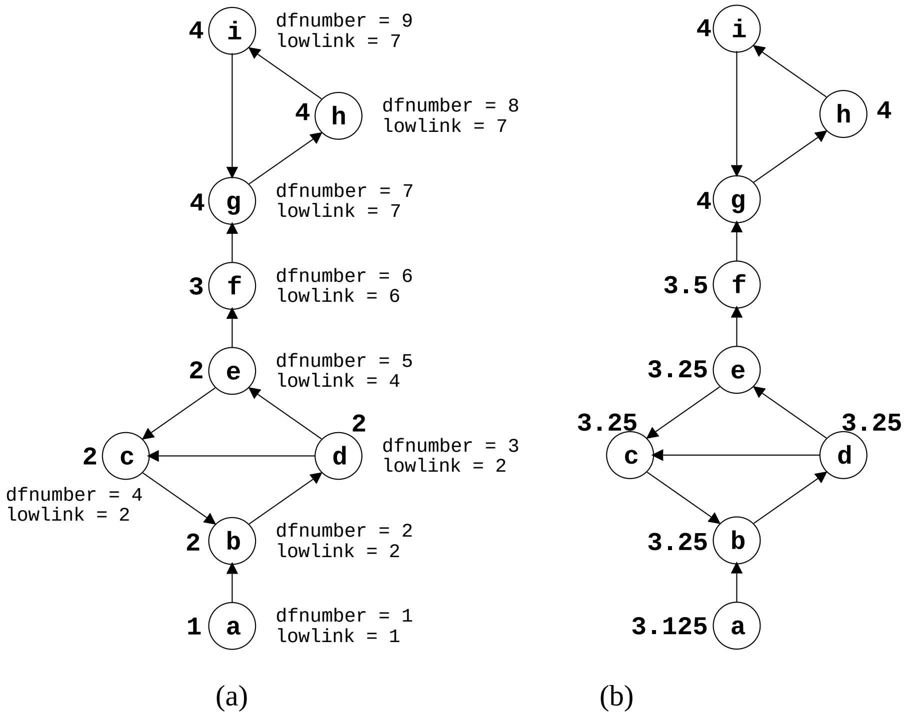


Figure 24: (a) The dfnumbers and lowlinks that would be computed for a sample dataflow graph by both the strong connectivity algorithm and `reorder`. (b) The position numbers that might be assigned to variables if `reorder` was called with `a` as the start variable and 3 as the position number.

information is computed in the false branch of the conditional on lines 12-16 of `reorder`. Readers may note that the conditional on line 11 of the strong connectivity algorithm that guarded this false branch is not included in the `reorder` algorithm. The reason this conditional can be deleted is that we know that the variable being revisited must be on the `reorder_stack`. If it were not on the `reorder_stack`, it would already have been assigned a new position number and would be a boundary node. If it were a boundary node, the conditional on line 11 of `reorder` would have prevented us from reaching the false branch on lines 15-16 of `reorder`. Thus we can simplify the code on line 11 of the strong connectivity algorithm that deals with visited variables by omitting the latter half of the condition which reads ‘‘**and** `w is on the stack`’’. In Figure 22, we have dropped the conditional altogether, since taking the lesser of the current variable’s lowlink and the visited variable’s dfnumber does no harm, even if the first part of the condition, `w.dfnumber < v.dfnumber`, is violated.

The final modification to the strong connectivity algorithm occurs in the section of code that enumerates strongly connected components. This section of code assigns position numbers to strongly connected components, and possibly marks the variables in the strongly connected component as out-of-date. As noted earlier, nodes in a cycle are evaluated using a nullification-reevaluation strategy. If the strongly connected component is a cycle, and if this cycle is the next ‘‘node’’ that the eager evaluator is going to evaluate, then the variables in the cycle should be marked out-of-date. The strongly connected component will be the next ‘‘node’’ evaluated if it is the last component enumerated by the strong connectivity algorithm, and if the position number which is assigned to it is less than or equal to the smallest position number on the evaluation queue. In Figure 24, the last strongly connected component enumerated consists only of the variable `a`, and thus no variables are marked out-of-date. However, if the search had begun at variable `b`, then the last strongly connected component enumerated would have been the cycle consisting of the variables `b`, `c`, `d`, and `e`. In this case, these variables would have been marked out-of-date if the position number that they were assigned, 3.25, was less than the position number of any variable on the evaluation queue.

The position numbers that are assigned to variables are determined in the same fashion in which they are determined when cycles are not allowed, with one exception. If the edge that is added to the dependency graph induces a cycle, then the position number of the variable that this edge emanates from (i.e., the demanded variable), can be used as the position number for the cycle. For example, in Figure 24.b, adding an edge from `f` to `a` induces a cycle, and thus the position number for `f` can be assigned to each node in the new cycle. `propagate` takes advantage of this reuse of position numbers when it calls `reorder_variables` to determine if a cycle still exists. It does so by pretending that a self edge has been created for a node in the cycle and calling `reorder_variables` with the node as both the source and destination of the edge. Since `reorder_variables` does not detect self circularities, mimicing the addition of an edge from a node to itself does not cause `reorder_variables` to find false cycles. However, it does cause `reorder_variables` to assign the same position number to the cycle, if the cycle still exists.

To illustrate how `reorder` works, suppose it is asked to renumber the dataflow graph in Figure 18, and that it is passed a position number of 3 and the variable `a`. Assume that all dependencies are up-to-date. Figure 24.a shows the dfnumbers and lowlinks that will be computed as `reorder` executes, while Figure 24.b shows the new position numbers that are computed. `reorder` stops at node `g`, since `g` is a boundary node (dfnumbers and lowlinks are shown for nodes `g`, `h`, and `i` because this figure is also used to show the dfnumbers and lowlinks that would be computed by the strong connectivity algorithm shown in Figure 23). Working back it assigns position number 3.5 to node `f`, and then detects the cycle consisting of `b`, `c`, `d`, and `e`. The only successor of this cycle is `f`, and `f`’s position number is propagated back from `e` to `d` and finally to the initial cycle node `b`. When the depth-first search finishes visiting `b`, it assigns a position number that is less than 3.5, 3.25, to the nodes in the cycle. The call to `reorder` does not induce a cycle, so `a` is given a position number between 3 and 3.25.

4.4.4 Example

Figure 25 illustrates how the eager evaluator handles cycles when the user changes the selection from object B to the label of the width cell ("87") in Figure 6. The change in selection causes `width_cell.value` to be reevaluated (Figure 25.a). As `width_cell.value` evaluates, it demands the value of `label`’s width. This causes a new dependency edge to be added from `label.width` to `width_cell.value` and creates the cycle shown in Figure 25.b. Since the two variables in the new dependency are out-of-order, `reorder_variables` is called and it updates the position numbers of the gray-shaded variables shown in Figure 25.c. It also marks these variables as out-of-date. The evaluation of width_cell.value is then aborted and width_cell.value is placed back on the evaluation queue. Since no other variables are on the queue, the eager evaluator immediately starts evaluating width_cell.value using a nullification/reevaluation strategy (note that it is still necessary to abort the first evaluation, since the evaluator must switch to cycle evaluation, which involves nullification/reevaluation). This results in the updating of width_cell.value, label.width and label.string as well as the associated dependencies (Figure 25.e). We assume that label.width changes and places its one successor, outside.width, that is not in label.width's cycle on the evaluation queue. outside.width is evaluated using normal evaluation, resulting in the final dependency graph shown in Figure 25.f.

4.4.5 Time Complexity

The time complexity of evaluating graphs with cycles is comparable to evaluating graphs without cycles. The two major differences between the two algorithms is in the assignment of position numbers and the evaluation of variables in a cycle. The procedure that assigns position numbers to variables when cycles are allowed uses a modified version of the depth-first search algorithm used to assign position numbers when cycles are not allowed. Thus the cost of updating position numbers with cycles is comparable to the cost of updating position numbers without cycles.

The eager evaluator treats each cycle as a node, so adding an edge to any variable in the cycle will cause all variables in the cycle to be reevaluated if the position numbers of the variables involved in the edge are out-of-order. In the worst case all of the variables in $\mathtt{affected}_i$ will be in the cycle, so up to ($|\mathtt{added\_edges}_i|\ |\mathtt{affected}_i| + |\mathtt{affected}_i|$) equations could be reevaluated. In practice, the average number of variables that must be reevaluated as the result of adding an edge is a small constant, and thus the average-case bounds on the time complexity of eager evaluation with cycles are the same as the bounds on the time complexity of eager evaluation without cycles.

4.4.6 Issues in Cycle Evaluation

There are a number of issues in cycle evaluation that need to be further discussed. First, the nullification/reevaluation strategy used to evaluate cycles goes around a cycle only once. This is guaranteed by setting variables' outofdate flags to false before evaluating them. If their value is requested a second time while the cycle is being evaluated, their old value will be returned. The advantage of this approach is that cycles are guaranteed to terminate (as long as the individual evaluation functions terminate). The disadvantage is that the cycle is not allowed to iterate to a fix point if one exists. Thus one or more constraints may be left unsatisfied.

Alternative methods could be used to try to avoid the problem of unsatisfied constraints. One possibility is to call a more powerful constraint solver and pass it the constraints in the cycle. Another possibility is to iterate around the cycle multiple times. This could be done by alternately marking the variables in the cycle out-of-date and evaluating them, until either the cycle converges to a fix point or until a specified number of iterations have been completed. If this strategy is employed, it is important to ensure that the cycle has not been broken during the last evaluation. If the cycle has been broken, then the elements of the cycle should have their position numbers updated and normal evaluation should be used.

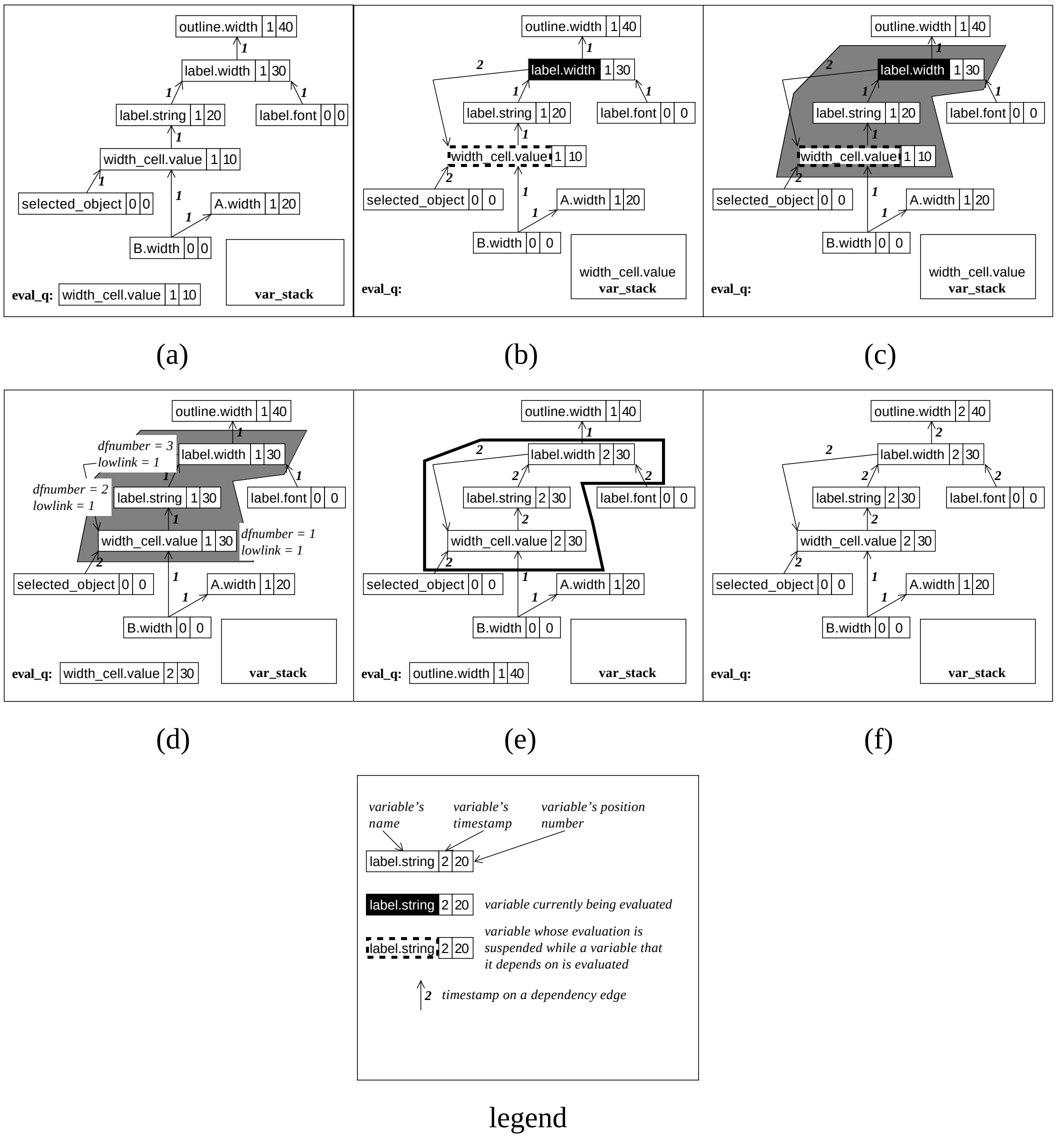
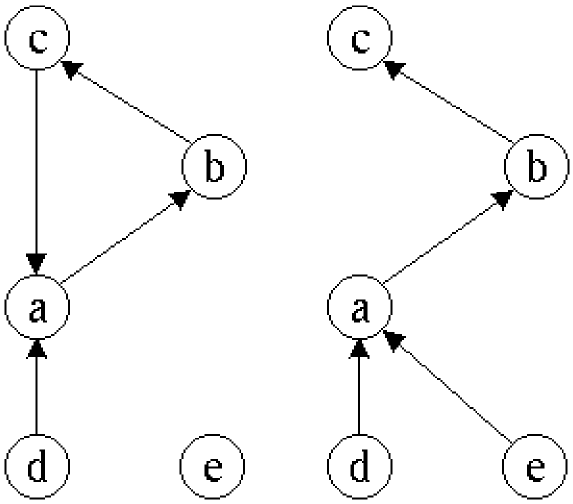


Figure 25: An example of the execution of the eager evaluation algorithm with cycles. (a) `width_cell.value` is placed on the evaluation queue when `selected_object` is changed to point at the width cell's label ("87") in Figure 6. (b) As `width_cell.value` is evaluated, it requests the value of `label.width`. This request creates a new dependency from `label.width` to `width_cell.value` and creates a cycle. The nodes involved in the cycle are shown inside the gray polygon in (c). (d) `reorder` recomputes the position numbers of the nodes in the cycle. `width_cell.value`'s evaluation is aborted and it is placed back on the evaluation queue. Since no other variables are on the queue, the eager evaluation starts evaluating `width_cell.value` again, except that this time it uses nullification/reevaluation and it evaluates all the nodes in the cycle. The nodes evaluated and the dependency edge timestamps that are updated during this phase are shown inside the polygon in (e). As a result of these evaluations, `outline.width` is placed on the evaluation queue. Once it is evaluated, the constraint system is up-to-date (f).

The test for the integrity of a cycle can be done efficiently by keeping track of which dependency edges are part of a cycle (this can be done by `reorder_variables`), examining the cycle edges before evaluating the cycle, and calling `reorder_variables` if one of the edges is stale. To simplify the presentation of the eager evaluator, this is not done in the cycle evaluator presented in this paper.

Care should be taken if a scheme other than nullification/reevaluation is used to evaluate a cycle, since the variables in the cycle should be evaluated in the correct order, even if the cycle is broken during cycle evaluation. For example, consider the cycle in Figure 26 and suppose that both variables `a` and `c` are on the evaluation queue. Further suppose that `d` is a pointer variable that used to point to `c` but now points to `e`. If the cycle is evaluated in an eager fashion (i.e., assume that all predecessors are up-to-date and add successors to the evaluation queue if the variable's value changes), then since `a` and `c` have the same position numbers, `c` may be evaluated first, which is incorrect. However, there is no way to know that `a` no longer depends on `c` before `a` is evaluated. Thus there is no way to prevent the eager evaluator from evaluating `c` first.



(a) (b)

Figure 26: (a) A cycle; (b) The cycle is broken when `a` is reevaluated and depends on a new variable `e`. The evaluator must ensure that a cycle is still intact before evaluating a cycle, or else variables may get evaluated out-of-order. For example, if the evaluator thinks the cycle is intact, it could evaluate `c` before `a`.

Finally, the eager evaluation algorithm avoids recording self-circularities. It does this by adding a test in `get_value` that checks whether a variable that is demanded by a formula is the same variable that the formula is trying to compute (in `get_value` the added test is `demanding_var` $\neq$ `v`).

Ignoring self-circularities is the simplest way to handle them since the variables will end up with the same value regardless of whether they are evaluated as cycle variables or normal variables. However, it is more efficient to evaluate them as normal variables. In addition, the strong connectivity algorithm used in `reorder` cannot detect self-circularities without special case code, so it is easier to avoid the problem by not creating the dependency.

4.5 Other Implementation Issues

Each time a constraint is evaluated, its value is cached so that the next time the constraint's value is requested, the constraint will not be reevaluated unless one of the variables it depends on has changed. Similarly the values of paths can be cached to improve efficiency. For example, in the labeled box example presented in Section 2.4, the label accessed the position of the box using the path `self.parent.box`. The first time this path is evaluated, the constraint solver can cache the resulting pointer to the box, so that as long as the variables comprising the path do not change, the constraint behaves as a direct reference constraint. The variables on this path still maintain dependency pointers to the constraint, so that if one of these variables changes, the path can be reevaluated and a new value cached for it.

Another optimization that has been used in Garnet is constant propagation [38]. The user is allowed to declare certain slots in an object to be constant. This information is then propagated by the constraint solver to other slots. If a constraint depends only on constant variables, the constraint solver computes the value of the constraint, stores the value in the slot, declares the slot constant, and throws away the constraint. This technique saves both time and space, since fewer constraints must be evaluated and the storage for discarded constraints can be reclaimed. For example, constant propagation eliminates 30 of the 43 constraints in the text button discussed in Section 1 and results in a space savings of 27% [38].

Another implementation issue that arises is what to do with constraints containing pointer variables that are unbound or which reference deleted objects. The two options considered in Garnet were 1) to destroy the constraint, keeping its previously computed value; or 2) to keep the constraint and return its previously computed value. Under option two, the constraint would again be evaluated once all its variables point at valid objects. We settled on the second option since, in many cases, the constraint will be used again. For example, a feedback object connected to an object that gets deleted should retain its constraints so that it can be connected to other objects.

5 Implementation Experience

Garnet currently uses lazy evaluation and a modified user-controlled version of caching that evaluates a path the first time the constraint is evaluated and then ignores it if the user assures the constraint solver that the path will never change. On an HP720 workstation running Lucid Common Lisp, an indirect reference to an object through a variable (e.g., `self.object_over.left`) requires roughly 182 microseconds, whereas a direct reference (e.g., `menu-item1.left`) or a reference that uses a cached path requires roughly 114 microseconds (the figures are approximate because they assume that the requesting variable is the only variable that depends on the demanded variable—each additional dependency adds 0.2 microseconds to the access cost). If a constraint does not have to be reevaluated, its previously computed value can be accessed in 9 microseconds, regardless of whether it is a direct reference or indirect reference constraint. Finally, a variable can be marked out-of-date in 70 microseconds. Garnet's constraint solver can solve indirect reference constraints quickly enough to allow feedback objects to track the mouse in real time or to perform smooth, realtime animations, even in large, constraint-based applications. For example, the Lapidary interactive design tool [32] consists of 16,000 lines of Lisp code and 16,700 constraints, all of which are indirect reference constraints, and is fast enough to provide instantaneous feedback to the user. The disparity between the number of lines and the number of constraints is explained by the fact that many of Lapidary's constraints are defined in the widgets used by Lapidary.

Profiles of various Garnet applications indicate that the constraint solver is efficient. For example, Table 1 shows the cumulative times required to update the display, execute the constraint methods, and perform the overhead required to maintain the formula and dependency graph data structures in a number of Garnet applications, including a game of Othello, a palette of menus, a palette of Motif widgets, and an arithmetic editor that allows box and wire diagrams of arithmetic expressions to be constructed. These times indicate that the constraint overhead is typically a relatively small fraction of the method execution time, which in turn is dominated by the time to update the display (in Othello, the method execution and constraint overhead times are roughly equal since the methods are unusually simple). The times for the constraint overhead are somewhat inflated by a number of features that Garnet offers (but which are not reflected in the algorithms presented in this paper), including constant checking, circularity checking for debugging purposes, dynamic type checking, and optional calls to ‘‘invalidate’’ and ‘‘pre-set’’ methods just before a slot is invalidated by `nullify` or assigned a new value in `get-value`.

Table 1: Cumulative number of seconds and percentage time spent updating the display, evaluating constraint methods, and maintaining the constraint system (e.g., adding/deleting constraints, invalidating variables, and updating/creating dependencies) for a number of benchmark applications.

	Update		Method Execution		Constraint Overhead	
Othello	6.6	82%	.7	9%	.76	9%
Menu Palette	10.3	88%	1.0	9%	.34	3%
Motif Widgets	14.5	75%	3.9	20%	1.0	5%
Arithmetic Editor	10.5	80%	1.7	13%	.95	7%

The current version of the eager evaluator requires 224 microseconds for a direct reference to an object and 441 microseconds for an indirect reference to an object (this is also on an HP720 workstation running Lucid Common Lisp). If a constraint does not have to be reevaluated, its previously computed value can be accessed in 12 microseconds. These times assume that the variable whose value is demanded has only one dependency edge. Each additional dependency edge adds 0.2 microseconds to the access cost. The difference in performance between the lazy and eager evaluators is to be expected since, in addition to evaluating the formulas and setting up dependencies, the eager evaluator must also compare position numbers, possibly do reorderings if the position numbers are out-of-order, generate new position numbers, manage a priority queue, and do circularity testing. However, since the eager evaluator does not have to do the bookkeeping of marking variables out-of-date, it may examine fewer equations than the lazy evaluator and thus take less time overall. In general, since a direct reference takes the eager evaluator 110 microseconds longer and an indirect reference takes the eager evaluator roughly 260 microseconds longer than the lazy evaluator, and since marking a variable out-of-date takes the lazy evaluator 70 microseconds, the lazy evaluator will tend to perform better than the eager evaluator unless the ratio of equations marked invalid to equations actually evaluated is well over 2 to 1. In most applications that we have tested Garnet on, the ratio of invalid equations to equations actually evaluated is under 2 to 1, so it is probable that the lazy algorithm will be retained as the default constraint solver. However, we are considering providing the eager evaluator as an alternative choice for application designers who believe that their applications would benefit from eager constraint solving.

6 Related Work

While pointer variables are commonly incorporated in programming languages, they have only recently been incorporated in their full generality in constraint systems. ThingLab [4] provides a limited form of indirect reference constraints. Designers can construct pathnames that allow a constraint to traverse a structure hierarchy to find an object. If one of the components in the structure hierarchy changes, the new object will be automatically referenced by the constraint. However, the constraint solving algorithm does not support arbitrary references to objects through pointer variables.

Coral also supported a restricted version of indirect reference constraints [48]. Coral allowed designers to declare the slots of an object that could be used as pointer variables for indirect reference constraints. Designers could then define constraints that accessed objects indirectly via these variables, like Garnet allows. However, the Coral pointer variables were not completely integrated into the constraint system. A special "set-variable" procedure was required to set the values of these variables. Programmers had to know whether a slot of an object was a pointer variable or not, and use the appropriate procedure to set it. In addition, the values of pointer variables could not themselves be defined using a constraint, thus restricting the applicability of the indirect reference constraints.

Penguims [21] and Eval/vite [23] support a model of indirect reference constraints that is somewhat more restricted than the one presented in this paper. The newer system, Eval/vite, allows constraints to be written in a limited subset of C++. Iteration is not yet supported, and constraints cannot have a variable number of inputs, which precludes writing constraints over dynamic sets of objects. Both of these restrictions may be lifted in the future. The restriction that constraints can only have a fixed number of input variables does lead to a more efficient implementation, because it is never necessary to dynamically add or remove edges from the graph. Since each variable has a fixed number of input edges, it is possible to simply adjust edges instead. For example, if a pointer variable causes a constraint to reference `B.left` rather than `A.left`, the incoming edge can be adjusted so that it originates from `B.left` rather than `A.left`.

Rendezvous [14, 15] supports indirect constraints for both the sources and targets of a constraint, permits both variable numbers of sources and targets, and allows constraints to consist of arbitrary Lisp expressions. In addition, Rendezvous provides support for side-effects and multi-output constraints, and prevents constraints from executing before all their inputs have been initialized. Rendezvous uses eager evaluation, but it differs in two respects from the algorithm presented in this paper. First, it does not use position numbers but instead uses depth-first search to visit and topologically order all the variables in $\texttt{affected}_i$. It then evaluates all the variables in $\texttt{affected}_i$. This approach may evaluate more than the minimum possible number of variables (because changes may quiesce), but it does not have the overhead of computing position numbers.

Second, Rendezvous takes a pessimistic approach to evaluating constraints whereas the eager algorithm presented in this paper takes an optimistic approach. Rendezvous will try not to evaluate a constraint until it is sure that all of the constraint's predecessors have been computed. Rendezvous knows the constraint's predecessors in advance because it uses a combination of programmer written source specifications and compiler inferencing to statically identify these predecessors. If there is any doubt as to whether a constraint should be evaluated, the constraint will be temporarily deferred. In contrast, Garnet does not determine a constraint's inputs until the constraint is evaluated. Thus there is no way to predict in advance if a constraint's position number is still valid, and the constraint is evaluated on the assumption that the position number is valid. If during the evaluation a new predecessor is computed which causes the constraint to become out-of-order, the evaluation will be terminated.

The absence of an input expression has the disadvantage of causing Garnet to prematurely start evaluating some variables. However, by dynamically detecting sources, Garnet can determine the exact set of sources used by a constraint on each invocation. In contrast, Rendezvous statically identifies the sources, which requires that all potential sources be listed. On any given constraint invocation, the set of sources actually used may be a subset of the sources listed. For example, if a constraint has a conditional of the form:

```
d.left = if (self.temperature > 0) then (b.left + 10) else (c.left + 10)
```

then the potential set of sources is the set `{self.temperature, b.left, c.left}`, but the actual set of sources is the set `{self.temperature, b.left}` if `self.temperature` is greater than 0, and `{self.temperature, c.left}` if `self.temperature` is less than or equal to 0. Consequently, the static specification of sources generates a worst-case dependency graph that may appear circular (when it is not) or otherwise unsolvable (when it is solvable). In practice, it appears that such worst-case dependency graphs may cause some loss in efficiency because constraints are unnecessarily reevaluated (e.g., a change to `c.left` will always cause the above constraint to be reevaluated), but that they do not create circular or unsolvable graphs that would not otherwise exist. The problem with overspecification can be solved by allowing programmer written specifications to be dynamically evaluated[5], although programmer written specifications introduce a potential source of errors that automatic source detection avoids.

Alphonse [17] is a program transformation system that takes a program annotated with Alphonse notations and converts it to an incremental program. Both caching and incremental graph evaluation algorithms are used to implement the programs. Like the indirect reference constraints in this paper, Alphonse supports pointer variables and dynamically changing sets of input variables. The description of Alphonse in [17] does not specify a specific algorithm that should be used for evaluating variables. However, it does describe how the dependency graph is dynamically maintained and its approach differs from the timestamp approach used in this paper. Alphonse maintains dependencies by removing all predecessor edges immediately before a formula is evaluated and then adding new edges during the current evaluation of a formula. While this approach is more efficient than using timestamps (with timestamps one must first determine whether there is a pre-existing edge), it cannot be used in our implementation because Garnet must be fault tolerant—if a formula crashes, the system must retain enough information to properly evaluate formulas once the appropriate variables are given new values. As pointed out in Section 4.2.1, if edges are removed from the dependency graph before it is certain that they are no longer needed, it is not possible to guarantee that formulas will be properly evaluated in the future if they crash.

Many other systems, such as Sketchpad [47], CONSTRAINTS [46], Grow [3], Apogee [20], Peridot [31], COOL [26], the Cornell Synthesizer Generator [41], MetaMouse [30], and CONSTRAINT [51], have used constraints but not pointer variables.

Pointer variables have also been explored in several somewhat different constraint contexts. Kaleidoscope [10] supports a different type of abstraction—constraint abstraction rather than procedural abstraction—in which procedures (called constraint constructors) consist of a set of constraint statements and produce as output a set of constraints instantiated with the parameters passed to the procedure. The constraints may contain indirect references, such as `rect.left = object_over.left`. Kaleidoscope has a well-defined notion of time and at each user-directed advance of time, `object_over` may be rebound. Internally, Kaleidoscope treats a rebinding as the retraction of one constraint (`rect.left = A.left` if `object_over` used to point to A) and the assertion of a new constraint (`rect.left = B.left` if `object_over` now points to B). Thus Kaleidoscope can satisfy constraints using an appropriate algorithm that handles direct references, such as DeltaBlue [11]. The pointer model presented in this paper differs from the Kaleidoscope model in that pointer variables are directly handled by the constraint satisfaction algorithms rather than by asserting and retracting constraints.

Pointer variables have also been examined in the context of logic-oriented languages, such as CLP [24], Prolog3 [7], and Concurrent Constraint Programming [44]. Pointer variables can be represented

[5]The Rendezvous constraint solver currently supports these programmer written specifications. The source expressions to be dynamically evaluated are specified by filling in code templates.

as unbound variables that can be subsequently unified with some object. The programmer can represent changeable state by asserting and retracting clauses and constraint relationships over time. These assertions and retractions allow the same constraints to be instantiated with different objects at different times.

Term rewriting systems provide yet another means for representing pointer variables within constraints. Bertrand [27], Siri [18, 19], and Equate [53] are several examples of such systems. Term rewriting systems bear a certain resemblance to logic languages in that the user writes a set of rewrite rules, each of which contains a pattern as a ‘‘head’’, and one or more expressions as a ‘‘body’’. These systems transform expressions by searching for subexpressions that match the patterns in the heads of the rewrite rules, and replacing them with the expressions in the body of the rewrite rules. Term rewriting systems are not as expressive as logic programs because they do not provide the full power of unification. However, just as in logic languages, a programmer can simulate pointer variables by adding and deleting rewrite rules over time.

Both the logic-oriented approaches and the term rewriting approaches differ from the Garnet approach in style and in the types of algorithms used to solve the constraints. Garnet represents pointer variables explicitly in the language and allows them to be directly manipulated by the user. The algorithms that Garnet uses also handle pointer variables directly. In contrast, the logic-oriented languages and the term rewriting systems represent pointer variables implicitly. Users cannot directly change the objects referenced by constraints. Rather, the old constraints are retracted and new constraints that reference the changed objects are added. The constraints themselves are still direct-reference constraints and the algorithms that satisfy them can only handle direct reference constraints.

There are many domain-dependent constraint satisfiers, such as simultaneous linear equation solvers [49, 27, 28], non-linear solvers [54, 8], and inequality solvers [25], that can solve more expressive constraints than one-way constraints, but they are less flexible, since they are typically limited to one domain, and they are slower on the subclass of constraints that one-way constraint satisfiers can solve. All of these solvers are also incapable of handling pointer variables.

Domain-independent, multi-way constraints are also more powerful than one-way constraints, but they can produce unexpected results and may fail to find a solution when one exists [23]. Both of these drawbacks arise from the need to first plan how to satisfy the constraints, before actually satisfying the constraints. Typically each constraint has multiple methods for solving it, and the plan selects one method for each constraint [11, 12, 52]. The methods are then executed in topological order, just as in one-way constraint solving. Thus the algorithms presented in this paper can be used in the evaluation phase of multi-way constraint satisfaction. The problem with manageability arises because the constraint solver may choose to solve a constraint in an unexpected way. For example, suppose the variable `right` is changed in the constraint `right = left + width` and the planner decides to change `left`. If the user expects it to change `width`, then the object will move instead of resizing, much to the user’s surprise. The notion of attaching priorities to constraints, called constraint hierarchies, has been advanced to help the user better manage this problem [6]. However, there are no widespread studies on their usefulness to programmers or on whether they fully solve the manageability problem. Until recently, multi-way solvers were considered too slow to incorporate in user interface systems, but this has changed with the introduction of fast incremental solvers, such as IncrementalPdof [52], DeltaBlue [11], and SkyBlue [42], and the use of optimization techniques such as constant propagation and plan caching [29, 9, 50]. However, each of these algorithms has drawbacks. IncrementalPdof does not incorporate a management scheme like constraint hierarchies, and DeltaBlue and SkyBlue are not designed to find solutions to a constraint graph with cycles[6]. Until these problems are solved to the satisfaction of programmers and users, one-way constraints are likely to remain more popular.

7 Future Research

There are several directions for future research. First, we are examining graphical means of tracing constraints so that designers can debug them more easily [34]. Another direction for future work is to develop multi-way indirect reference constraint systems. We have a design for a two-way indirect reference constraint system, based on the ideas in this paper. It is also be possible to build multi-way constraint systems that support indirect references, but which internally use direct reference algorithms. MultiGarnet, developed by Michael Sannella, is an experimental version of Garnet which uses such a scheme. MultiGarnet uses a direct reference algorithm called SkyBlue [42] that extends the DeltaBlue multi-way constraint algorithm [11] to handle multi-output constraints and to more gracefully handle cycles. MultiGarnet currently handles changes to indirect references by removing the changed constraint, and re-adding it with the updated reference. SkyBlue is designed to add and remove constraints quickly, but this particular situation might be handled more efficiently using algorithms such as those described in this paper.

8 Conclusions

Indirect reference constraints significantly extend the potential uses of constraints in interactive applications by allowing constraints to express the dynamic behavior that occurs inside an application's window. These constraints can be used to specify animations and feedback that operate over dynamic sets of objects, implement copying and instancing of structured objects in prototype-instance systems, simplify the creation of prototype objects from example objects in demonstrational systems, and abstractly specify layouts. Indirect reference constraints also open the way for constraints to be used in maintaining pointer-based data structures, such as lists, trees, and graphs. In addition, their programming style is simpler and more effective than conventional constraints, they improve the efficiency of applications, and they decrease an application's storage demands. Because of their flexibility and ease of use, indirect reference constraints permitted a comprehensive user interface toolkit, Garnet, to be built on top of the constraint system (in previous toolkits, constraints had been built in at a higher level and the low level parts of the toolkits were not implemented using constraints). Many applications with thousands of indirect reference constraints have been built using Garnet, thus demonstrating the usefulness of these constraints. The development of indirect reference constraints represents an important step toward the

[6]A constraint graph is undirected—finding a solution involves directing the edges to form a directed, acyclic dependency graph. Typically, multiple dependency graphs can be extracted from a constraint graph, depending on how the edges are directed. If one of these dependency graphs is cyclic, then the constraint graph is said to be cyclic.

development of a general-purpose, constraint-based, interactive programming language.

Acknowledgements

Michael Sannella has developed the experimental multi-way system described in the future work section and we appreciate his assistance in helping us identify future directions for research on pointer variables. The comments of the referees were very helpful in improving the organization and clarity of the paper's presentation. Development of Garnet is partially supported by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597. The work of B. Vander Zanden was also supported in part by NSF grant IRI-9111121.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

References

1. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts, 1974.

2. Alduss. Alduss Intellidraw.

3. Paul Barth. "An Object-Oriented Approach to Graphical Interfaces". *ACM Transactions on Graphics 5*, 2 (April 1986), 142-172.

4. Alan Borning. "The Programming Language Aspects of ThingLab; a Constraint-Oriented Simulation Laboratory". *ACM Transactions on Programming Languages and Systems 3*, 4 (Oct. 1981), 353-387.

5. Alan Borning and Robert Duisberg. "Constraint-Based Tools for Building User Interfaces". *ACM Transactions on Graphics 5*, 4 (Oct. 1986), 345-374.

6. A. Borning, R. Duisberg, B. Freeman-Benson, A. Kramer, M. Woolf. Constraint Hierarchies. OOPSLA'87 Conference Proceedings, 1987, pp. 48-60.

7. Jacques Cohen. "Constraint Logic Programming Languages". *Communications of the ACM 33*, 7 (July 1990), 52-68.

8. J.E. Dennis, Jr., and R.B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* Prentice Hall, New York, NY, 1983.

9. Bjorn N. Freeman-Benson. "A Module Mechanism for Constraints in Smalltalk". *Sigplan Notices 24*, 10 (Oct. 1989), 389-396. ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'89.

10. Bjorn N. Freeman-Benson. Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming. OOPSLA/ECOOP'90 Conference Proceedings, 1990, pp. 77-88.

11. Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. "An Incremental Constraint Solver". *Comm. ACM 33*, 1 (Jan. 1990), 54-63.

12. Jim Gosling. Algebraic Constraints. Tech. Rept. CMU-CS-83-132, Carnegie Mellon University Computer Science Department, 1983.

13. Tyson R. Henry and Scott E. Hudson. Using Active Data in a UIMS. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'88, Banff, Alberta, Canada, Oct., 1988, pp. 167-178.

14. Ralph D. Hill. Languages for the Construction of Multi-User Multi-Media Synchronous (MUMMS) Applications. In Brad A. Myers, Ed., *Languages for Developing User Interfaces*, Jones and Bartlett Publishers, Boston, MA, 1992, pp. 125-143.

15. Ralph D. Hill. The Rendezvous Constraint Maintenance System. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'93, Atlanta, GA, Nov., 1993, pp. 225-234.

16. R. Hoover. *Incremental Graph Evaluation*. Ph.D. Th., Department of Computer Science, Cornell University, Ithaca, NY, 1987.

17. Roger Hoover. "Alphonse: Incremental Computation as a Programming Abstraction". *Sigplan Notices 27*, 7 (July 1992), 261-272. ACM SIGPLAN'92 Conference on Programming Language Design and Implementation.

18. Bruce Horn. Properties of User Interface Systems and the Siri Programming Language. In Brad A. Myers, Ed., *Languages for Developing User Interfaces*, Jones and Bartlett Publishers, Boston, MA, 1992, pp. 211-238.

19. Bruce Horn. "Constraint Patterns As a Basis For Object Oriented Programming". *Sigplan Notices 27*, 10 (Oct. 1992), 218-233. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications; OOPSLA'92.

20. Scott E. Hudson. Graphical Specification of Flexible User Interface Displays. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'89, Williamsburg, VA, Nov., 1989, pp. 105-114.

21. Scott E. Hudson. An Enhanced Spreadsheet Model for User Interface Specification. Tech. Rept. TR90-33, The University of Arizona, 1990.

22. Scott E. Hudson. "Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update". *ACM TOPLAS 13*, 3 (July 1991), 315-341.

23. Scott E. Hudson. A System for Efficient and Flexible One-Way Constraint Evaluation in C++. Tech. Rept. 93-15, Graphics Visualizaton and Usability Center, College of Computing, Georgia Institute of Technology, April, 1993.

24. J. Jaffar and J. Lassez. Constraint Logic Programming. Proceedings of the Principles of Programming Languages Conference, Munich, Germany, Jan., 1987, pp. 111-119.

25. J. Jaffar, S. Michaylov, P.J. Stuckey, and R.H.C. Yap. "The CLP(R) Language and System". *ACM TOPLAS 14*, 3 (July 1992), 339-395.

26. Tomihisa Kamada and Satoru Kawai. "A General Framework for Visualizing Abstract Objects and Relations". *ACM Transactions on Graphics 10,* 1 (Jan. 1990), 1-39.

27. W. Leler. *Constraint Programming Languages: Their Specification and Generation.* Addison-Wesley Publishing Company, New York, 1988.

28. J. Li. Constraint Hierarchies as Triangular Systems. Tech. Rept. TRITA-NA-P9130, Dept of Numerical Analysis and Computing Science, Royal Institute of Technology, S-100 44 Stockholm, Sweden, 1991.

29. John Maloney, Alan Borning, and Bjorn Freeman-Benson. "Constraint Technology for User-Interface Construction in ThingLabII". *Sigplan Notices 24*, 10 (Oct. 1989), 381-388. ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'89.

30. David L. Maulsby, Ian H. Witten, and Kenneth A. Kittlitz. Metamouse: Specifying Graphical Procedures by Example. Computer Graphics, Proceedings SIGGRAPH'89, Boston, MA, July, 1989, pp. 127-136.

31. Brad A. Myers. *Creating User Interfaces by Demonstration.* Academic Press, Boston, 1988.

32. Brad A. Myers, Brad Vander Zanden, and Roger B. Dannenberg. Creating Graphical Interactive Application Objects by Demonstration. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'89, Williamsburg, VA, Nov., 1989, pp. 95-104.

33. Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Ed Pervin, Andrew Mickish, and Philippe Marchal. "Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces". *IEEE Computer 23*, 11 (Nov. 1990), 71-85.

34. Brad A. Myers. Graphical Techniques in a Spreadsheet for Specifying User Interfaces. Human Factors in Computing Systems, Proceedings SIGCHI'91, New Orleans, LA, April, 1991, pp. 243-249.

35. Brad A. Myers and Brad Vander Zanden. "An Environment for Rapid Creation of Interactive Design Tools". *The Visual Computer; International Journal of Computer Graphics 8*, 3 (1992), 94-116.

36. Brad A. Myers, Dario A. Giuse, and Brad Vander Zanden. "Declarative Programming in a Prototype-Instance System: Object-Oriented Programming Without Writing Methods". *Sigplan Notices 27*, 10 (Oct. 1992), 184-200. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications; OOPSLA'92.

37. Brad A. Myers. "Demonstrational Interfaces: A Step Beyond Direct Manipulation". *IEEE Computer 25*, 8 (Aug. 1992), 61-73.

38. Brad Myers, Dario A. Giuse, Andrew Mickish, and David Kosbie. Making Structured Graphics and Constraints Practical for Large-Scale Applications. Submitted for Publication.

39. Greg Nelson. Juno, a Constraint-Based Graphics System. Computer Graphics, Proceedings SIGGRAPH'85, San Francisco, CA, July, 1985, pp. 235-243.

40. T. Reps, T. Teitelbaum, and A. Demers. "Incremental Context-Dependent Analysis for Language-Based Editors". *ACM TOPLAS 5*, 3 (July 1983), 449-477.

41. T. Reps and T. Teitelbaum. *The Synthesizer Generator.* Springer-Verlag, New York, 1988.

42. Michael Sannella. The SkyBlue Constraint Solver. Tech. Rept. 92-07-02, Computer Science Department, University of Washington, July, 1992.

43. Michael Sannella and Alan Borning. Multi-Garnet: Integrating Multi-Way Constraints with Garnet. Tech. Rept. 92-07-01, Department of Computer Science and Engineering, University of Washington, Sept., 1992.

44. V. A. Saraswat. *Concurrent Constraint Programming Languages*. Ph.D. Th., School of Computer Science, CMU, Pittsburgh, PA, 1989.

45. Daniel D. Sleator and Paul F. Dietz. Two Algorithms for Maintaining Order in a List. Tech. Rept. CMU-CS-88-113, Carnegie Mellon University, September, 1988.

46. Guy L. Steele, Jr. *The Definition and Implementation of A Computer Programming Language based on Constraints*. Ph.D. Th., Department of Computer Science, MIT, Boston, MA, 1980.

47. Ivan E. Sutherland. Sketchpad: A Man-Machine Graphical Communication System. AFIPS Spring Joint Computer Conference, 1963, pp. 329-346.

48. Pedro A. Szekely and Brad A. Myers. "A User Interface Toolkit Based on Graphical Objects and Constraints". *Sigplan Notices 23*, 11 (Nov. 1988), 36-45. ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'88.

49. Christopher J. Van Wyk. "A High-level Language for Specifying Pictures". *ACM Transactions on Graphics 1*, 2 (April 1982), 163-182.

50. Brad T. Vander Zanden. *Incremental Constraint Satisfaction and Its Application to Graphical Interfaces*. Ph.D. Th., Cornell University, Ithaca, NY, 1988.

51. Brad T. Vander Zanden. Constraint Grammars--A New Model for Specifying Graphical Applications. Human Factors in Computing Systems, Proceedings SIGCHI'89, Austin, TX, April, 1989, pp. 325-330.

52. Brad Vander Zanden. A Domain-Independent Algorithm for Incrementally Satisfying Multi-Way Constraints. Tech. Rept. CS-92-160, University of Tennessee, July, 1992.

53. Michael R. Wilk. "Equate: An Object-Oriented Constraint Solver". *Sigplan Notices 26*, 11 (Nov. 1991), 286-298. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications; OOPSLA'91.

54. A. Witkin and W. Welch. Fast Animation and Control of Nonrigid Structures. Computer Graphics: SIGGRAPH'90 Conference Proceedings, Aug., 1990, pp. 243-252.

Table of Contents

List of Figures

List of Tables