Constraints and Object Identity

Gus Lopez[1], Bjørn Freeman-Benson[2], and Alan Borning[1]

[1]University of Washington, Dept. of Computer Science & Engineering, FR-35, University of Washington, Seattle, WA 98195, USA, {lopez,borning}@cs.washington.edu
[2]Århus University, Århus, Denmark. Current address: Carleton University, School of Computer Science, 514 Herzberg Building, 1125 Colonel By Drive, Ottawa, Ontario, Canada, K1S 0G9, bnfb@scs.carleton.ca

Abstract. Constraint imperative programming is an integration of declarative constraints and imperative object-oriented programming. The primary goal of this integration is to use constraints to express relations among objects explicitly—relations that were implicit in the code in previous languages. However, one of the fundamental concepts of object-oriented programming, object identity, can result in implicit relations, even when explicit identity constraints are supported. We analyze the problem and propose a solution—identity constraints—which we have implemented in our Kaleidoscope'93 language. This solution is understandable, efficiently implementable, and compatible with the Kaleidoscope constraint model.

Keywords. object identity, constraints, constraint imperative programming, Kaleidoscope, aliasing

1 Introduction

Object identity is a fundamental and essential concept in object-oriented languages for at least two reasons. First, object-oriented languages are often used to model aspects of the real world.[1] Suppose we have an object that models some phenomenon, e.g., a train reservation card (Figure 1a). A program in which two variables local and express refer to this single object (Figure 1b) models a very different situation from one in which local and express refer to different but equal objects. Second, object identity is important to the programmer, and plays a key role in many programming idioms in imperative object-oriented languages. If alpha and beta refer to the same object, a change to alpha is immediately visible in beta (Figure 1c); whereas if alpha and beta refer to different but equal objects, a change to alpha does not affect beta (Figure 1d).

Explicit relations are a fundamental concept in declarative constraint programming languages. Such languages are designed such that the behavior of a program is independent of the underlying implementation strategy, and more specifically, independent of the order in which the constraints are solved.[2] This independence is achieved by

1. This perspective on object-oriented programming has been articulated strongly by the designers of Simula [Dahl & Nygaard 66, Krogdahl & Olsen 86] and Beta [Madsen et al. 93].

2. See [Freeman-Benson et al. 90] or [Leler 87] for an overview of constraint systems and languages.

requiring all relations, including any ordering or sequencing relations, to be stated explicitly as constraints.

Because a constraint imperative programming (CIP) language, such as Kaleidoscope'93 [Lopez et al. 93], is at once both a constraint language and an imperative object-oriented language, it should include the fundamental and essential concepts of each. Thus, a CIP language must both have object identity, and ensure that all relations are explicit.

In previous work [Freeman-Benson 91, Lopez et al. 93], we described the constraint imperative programming framework, and successive designs and implementations of CIP languages: Kaleidoscope'90, Kaleidoscope'91, and Kaleidoscope'93. In the CIP framework, constraints can have various durations, including *always* (meaning that the constraint should be enforced throughout the remaining duration of execution), *once* (meaning that the constraint should be enforced at that instant, but then removed), and *during* (meaning that the constraint should be enforced for the duration of a particular loop; usually for the duration of a user interaction). To reconcile imperative state change with declarative constraints, an assignment statement, such as `x := x+5`, is regarded as a once constraint relating successive states of its variables.

In the CIP framework, equality constraints (which allow objects to maintain separate identities but to have equal slot values) can replace many common uses of identity from conventional object-oriented languages. This replacement helps to guard against unintended side effects, e.g., Figure 1(c & d). The Kaleidoscope'90 design viewed identity as an implementation technique for improving efficiency, but not as a semantically significant language element. We now regard this as incorrect: identity is significant both for modelling the real world and for use in many practical programming idioms (for example, traversing linked lists or building cyclic graphs).

At first glance, it would seem that overcoming the "CIP Identity Gap" is easily done by adding conventional constructs for object identity, such as pointers. However, in a CIP language, there are a number of subtle interactions between object identity and constraints. In the remainder of this paper, we explore these interactions, list goals for a

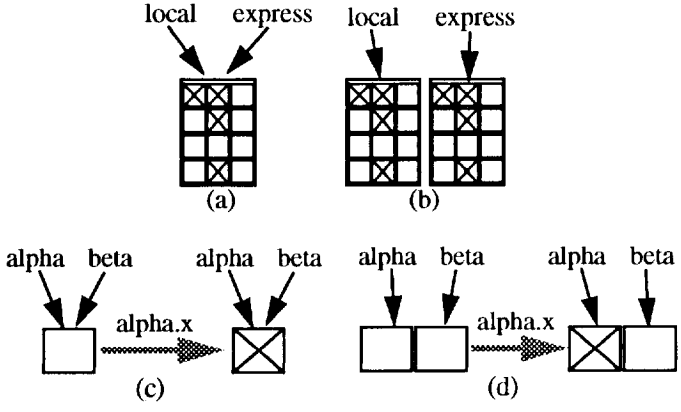


FIGURE 1. Uses of identity

reconciliation of object identity with constraints, and describe our solution from Kaleidoscope'93. We also discuss a number of plausible, but less acceptable, alternative designs.

2 Goals

As noted previously, in our earlier work on constraint imperative programming, we treated object identity as a technique for improving the efficiency of the implementation, but not as a part of the language semantics. Other CIP languages, e.g., Siri [Horn 92, Horn 93], have a traditional notion of object identity, but restrict constraints to operate within the bounds of their defining object. As described in the introduction, we now believe that a CIP language should fully support a notion of object identity in its semantics. We therefore propose the following goals for combining object identity and constraints:

1. **Mutable State.** The language should support objects with mutable state.
2. **Constrained Values.** The language should support constraints over object values. (An expanded list of goals for such constraints is given in a previous ECOOP paper [Freeman-Benson & Borning 92]; all of those goals remain important for our current design efforts.)
3. **Object Identity.** The language semantics should include object identity in addition to object equality.
4. **Identity Constraints.** Specifying that a variable should refer to a specific object should be done using constraints. Insofar as possible, these identity constraints should be analogous to value constraints, and should be integrated smoothly with them.
5. **Evolution.** The object identity model should seem reasonable to a user of a standard object-oriented language, and should be a natural evolutionary step.
6. **Efficiency.** The constructs for identity should be efficiently implementable. If a program uses no constraints beyond the analog of standard assignment statements, then it should be possible to compile the source code to be competitive with standard object-oriented languages.

In conjunction with these goals, we have identified a number of core issues faced by designs for identity in constraint imperative languages:

- Identity mechanisms permit multiple variables to refer to the same object. How is this expressed and controlled?
- Is it by conventional assignment statements or by identity constraints?
- If by identity constraints, can identity constraints be one-way, multi-way, or both?
- If by identity constraints, are weak identity constraints allowed?
- What relation is used to retain values for objects in the absence of stronger constraints to change these values? Equality? Identity?

- How are the durations of identity relationships expressed?
- When an identity relation become inactive, what happens to the identity of previously constrained objects?
- In what situations can unconstrained aliasing arise?

3 Equality versus Identity

Identity and equality are similar in many respects, and thus are often confused or misused in a language that does not sufficiently support both. One reason for this is obvious: if two values are identical, then they are clearly equal. Thus, in a language with support for identity but not for explicit equality constraints, the most convenient mechanism for ensuring that two values are equal is to make them identical. (In other words, if variables x and y should refer to equal values, we achieve this by aliasing x and y.) C++ and Smalltalk fall into this category of languages. Their support for identity is via pointers and pointer assignment.

If a language with adequate support for explicit equality relations is available (e.g., a CIP language), then equality is often a better choice than aliasing. For example, suppose that we have a CAD system for designing railroad cars and are using it to design two similar funicular cars. The cars should have the same seating layout, but they are not identical: one has right-hand side doors, the other left-hand side. The following

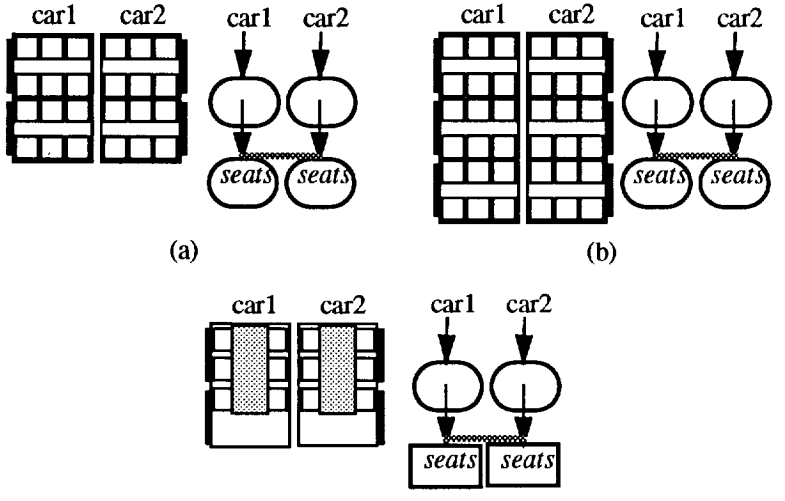


FIGURE 2. Equal but not identical

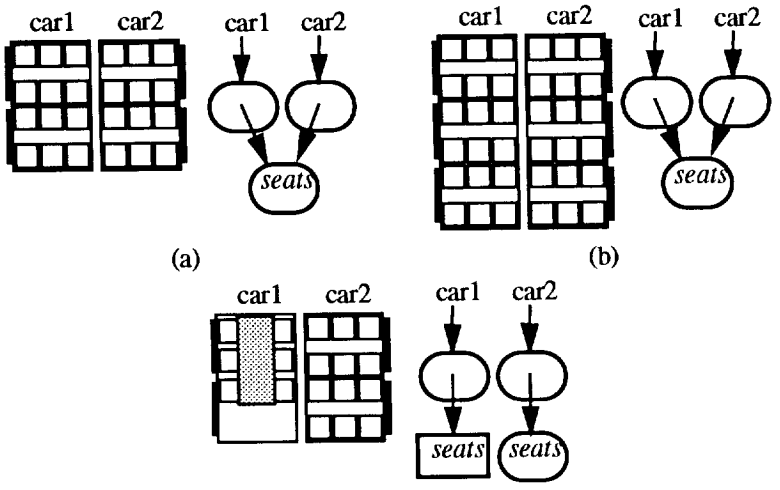


FIGURE 3. Identical but only implicitly equal

code reflects this situation. (See also Figure 2a, in which the grey line indicates the equality constraint).

```
car1.seating = car2.seating; /* An enforced equality relation, not an equality test. */
car1.doors := left;
car2.doors := right;
```

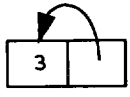
The two cars are modelled correctly as distinct objects, and the explicit equality constraint between their seating layouts ensures that any changes to one will be automatically reflected in the other (Figure 2b). Even if car1 were to change, e.g., through assignment, the change would still be reflected in car2 (Figure 2c):

```
car1.seating := presidential_seating;
```

In contrast, if car1.seating and car2.seating were aliased, they would also be equal, but their equality would be implicit rather than explicit. Furthermore, this implicit constraint could be broken by an action as innocuous as an assignment (Figure 3(a,b,c)).

However, equality alone is not sufficient for an object-oriented language. First, it does not handle the systems modelling aspect of object identity. Second, it does not handle many common programming idioms used in object-oriented programs. To illustrate this second point in a simple context, consider the problem of specifying a circular list. The desired structure is easy to specify in an imperative language with pointers. For example, in Common Lisp:

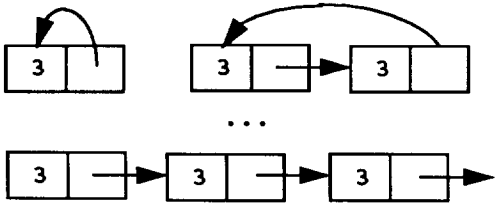
```
(setf a (list 3))
(setf (rest a) a)
```



Using constraints in Kaleidoscope, we might try:

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

However, because the constraints only imply equality and not identity, they could be satisfied by an infinite number of different graph structures, including a non-circular, infinite list:



One might try adding a minimality condition to the constraint solver such that it returned the solution with the smallest number of cons cells—but what if a two cell list was what we really wanted? Even if a one-cell list was wanted, given the importance of object identity for system modelling, we would like to give the programmer more explicit control. Identity is clearly a better solution here. With identity constraints, the single-cell circular list can be specified as:

```
a.head = 3;                 /* An enforced equality constraint. */
a.tail == a;                /* An enforced identity constraint. */
```

4 Our Design

The basic constraint that two variables x and y be identical is written x == y. A number of annotations are available in Kaleidoscope for value constraints: namely, read-only annotations on variables, strengths on constraints, and the constraint durations once, during, and always. A read-only annotation on a variable indicates that constraint may not affect the value of that variable, so that operationally, the satisfier must change another variable or variables instead to satisfy the constraint. (A declarative semantics for both read-only annotations and strengths is given in [Borning et al. 92].) A strength of required indicates that the constraint must be satisfied, while other strengths indicate that the constraint should be satisfied if possible, but that it is not an error if it cannot. Finally, as described previously, durations indicate the period during which the constraint is in force. All of these annotations are available for identity constraints as well. For example, once: x==y indicates that the identity constraint should be enforced, and immediately afterward the constraint is no longer active, while always: x==y? is always active, but can only be satisfied by changing the identity of x, not that of y.

The assignment statement for values v := expr in Kaleidoscope is equivalent to the following pair of constraints:

```
once: temp = expr?;
once: v = temp?;
```

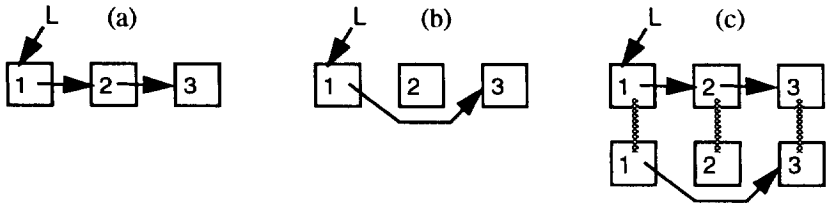


FIGURE 4. Parameter passing: equality vs. identity

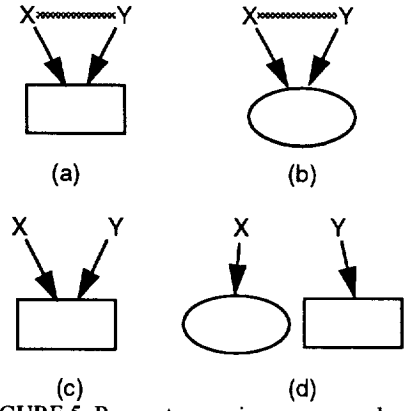


FIGURE 5. Parameter passing: once vs. always

For example, x := x+5 results in evaluating x+5, storing the result in temp (since x+5 is annotated as read-only, changing temp is the only way of satisfying this constraint), and then storing the value of temp back in x. A similar shorthand for identity-based assignment is available as well: v :== expr, which is equivalent to:

```
once: temp == expr?;
once: v == temp?;
```

4.1 Multi-Way vs. One-Way; Once vs. Always

All four kinds of identity constraints are essential in our design: one-way and multi-way; once and always. Parameter passing uses multi-way always identity, type invariants (see Section 4.5) use both types of always identity, and common programming idioms use one-way once identity.

Parameter Passing. Procedures in Kaleidoscope are similar to methods in traditional object-oriented languages. Constraint constructors are special procedures that define the meanings of non-primitive constraints. For example, here is the + constructor for cartesian points:

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

Parameters to procedures and constraint constructors are passed "call-by-identity" using multi-way always identity constraints:

- Identity rather than equality. If equality were used, then statements manipulating the identity of objects could not be encapsulated in a procedure. As an example, Figure 4 illustrates the list data structure L before the following procedure (a), after using identity parameter passing (b), and after using equality parameter passing (c).

```
procedure remove_one(d)
  d.next :== d.next.next;
end procedure remove_one;
remove_one(L);
```

- Multi-way rather than one-way. Otherwise, constraint constructors would be restricted to use some of their arguments as inputs and some as outputs, contrary to the normal multi-way spirit of constraints. Consider the following example:

```
constructor celsius_to_fahrenheit(in c, out f)
   c * 1.8 = f - 32.0;
end constructor celsius_to_fahrenheit;
celsius_to_fahrenheit( danish, american );
danish := 8;                    /* works correctly: danish → c =...= f → american */
american := 56;                 /* does not work because american cannot flow into f */
```

- Always rather than once. This allows other always constraints created during procedure execution have effect beyond the life of the procedure. As an example, Figure 5 illustrates the data structures when always or once identity constraints are used to pass parameters to make_the_same:

```
/* assume x and y contain Rectangles */
procedure make_the_same(a,b)
   always: a == b;
end procedure make_the_same;
make_the_same(x,y);          /* Figure 5a (always) and Figure 5c (once) */
x :== new Oval;              /* Figure 5b (always) and Figure 5d (once) */
```

Programming Idioms. For many common programming idioms, one-way, rather than multi-way, identity constraints are essential. For example, consider the following code to redisplay the active windows by iterating through a list. Because each Window object is a proxy for an operating system data structure, the system should use those objects, not equal copies of them.[3]

```
w :== Screen.ActiveWindows;
while w ≠ nil do
  display(w.head);
   w :== w.tail;
end while;
```

3. If objects have termination routines, such as C++ destructors, then creating and destroying equal copies of proxy objects can have harmful side-effects. For example, when the copy is destroyed, its termination routine may release the external resources while other copies of the proxy object are still active. Even if the program would execute correctly with equal copies rather than the single window, we believe that programmers would like to have control over this aspect of their program (Goal 5 — Constraint imperative programming as an evolutionary advance on current object-oriented programming languages).

As described above, the two assignments, w :== Screen.ActiveWindows and w :== w.tail, are equivalent to once one-way identity constraints. Clearly we want these to be once rather than always constraints — otherwise w would be permanently bound to the entire list of active windows, and could not march down the list. We further want one-way rather than multi-way constraints here (for example, we want to make sure that the assignment statement w :== Screen.ActiveWindows results in changing w rather than Screen.ActiveWindows).

Other Uses. Supplying the same annotations and durations for identity constraints as for value constraints allows identity constraints to fit neatly in the familiar CIP constraint model. In addition, specifying identity with constraints gives additional power beyond that available in a more traditional language: for example, in a debugging application, one can use an always identity constraint to track the identity of the object referred to by some variable.

4.2 Stay Constraints

As described in the introduction, to reconcile imperative state change with declarative constraints, we may view a variable as consisting of a stream of successive states. In this view, an assignment statement, e.g., x := x+5, is a once constraint relating successive states of its variables. Additionally, to ensure that variables remain the same over time in the absence of some stronger influence, the language automatically provides very weak stay constraints between the successive states. In Kaleidoscope'93, these stay constraints are weak identity constraints, so that a variable will continue to point to an object with the same unique ID as time advances.

In contrast, in our original Kaleidoscope'90 design, stay constraints were equality constraints, so that there was no way to talk in the language semantics about the identity of an object through time. In addition, in Kaleidoscope'90, variables literally were represented as streams of values, and assignments were implemented as constraints (resulting in a correct but very slow implementation).

The shift to stay-as-identity in Kaleidoscope'93 has allowed us to make a major optimization in our implementation: namely, variables are implemented in a conventional fashion, as locations in memory that refer to a block of storage representing a given object. While semantically an assignment statement remains a constraint between successive states of a variable, the implementation need store only one state. As the program executes, it perturbs the values of certain objects, resulting in the activation of the constraint satisfier, which then returns the object store to a consistent state.[4]

4.3 Unconstrained Aliasing

Constraints are enforced only while active, i.e., as soon as the constraint becomes inactive, it has no further effect. Thus a once constraint is only momentarily active, and an assert-during constraint is only active during the execution of the associated block of code. But, while the effect of an inactive value constraint is gone, its result may remain, and the relation may happen to continue to hold. For example, suppose we

have a constraint A+B=C. While the constraint is active, the values in the A, B, and C variables have a plus relation, say, 4, 5, and 9. When the A+B=C constraint is removed, the values may remain in a plus relation, but there is no further enforcement of the plus constraint. (For example, if C were subsequently changed, there would be no change in A or B if there were no other constraints relating them.)

In contrast, in the case of an identity constraint, this accidental satisfaction of the constraint beyond its declared duration *can* have an effect. Consider the situation when alpha and beta are constrained to be identical (i.e., they are aliased) and then the identity constraint is removed. While they are constrained, they will obviously refer to the same object. When the constraint is removed, they will continue to refer to the same object until one or the other is re-directed, since identity is used for stay constraints. We term this situation *unconstrained aliasing*. (In the compiler literature, this phenomenon is termed *accidental aliasing*, but we use the term unconstrained aliasing instead since the aliasing may be deliberate even though unconstrained, e.g., the Programming Idioms example previously.) Obviously, always identity constraints cannot give rise to unconstrained aliasing and thus should be used when possible.

As an illustration of the problems that can arise from unconstrained aliasing, compare

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

with

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

In the first example, the once identity constraint still has an effect after its duration, since a and b remain identical, so that b.x is also set to 5. In the second example, the once equality constraint has no effect after it is removed, and b.x remains 0.

In many cases, however, this unconstrained identity is the desired result. For example, consider again the code from Section 4.1 that redisplays a list of active windows by iterating through them. (The code is shown again at the top of the next page.)

---

4. The stream model of Kaleidoscope'90 is an instance of the refinement model for constraint systems. In this model, constraints are accumulated as the computation progresses. There is never a notion of retracting a value, only of accumulating additional information and refining the permissible values of variables. In contrast, the Kaleidoscope'93 implementation uses the perturbation model for constraint systems. In this latter model, at the beginning of a computation step there is a state of the system, which satisfies all the required constraints, and which satisfies the preferential constraints as well as possible. The system is perturbed, for example by an assignment or by a user input event; and the constraints are then re-satisfied. The refinement and perturbation models are compared at greater length in [Borning et al. 92].

```
w :== Screen.ActiveWindows;
while w ≠ nil do
  display(w.head);
  w :== w.tail;
end while;
```

The two assignments, w :== Screen.ActiveWindows and w :== w.tail, are equivalent to once one-way identity constraints. They change the identity of w to elements of the list, but the use of that element is after the once constraint has become inactive. In other words, this program relies on the unconstrained aliasing of two variables.

4.4 Report Card

In Section 2 we listed a number of goals for combining object identity and constraints. Goals 1, 2, and 3 (support for mutable state, constrained values, and object identity) are clearly achieved by the design. Goal 4 stated that specifying that a variable refer to a specific object should be done using constraints, and that insofar as possible, these identity constraints should be analogous to value constraints, and integrated smoothly with them. We believe that this goal has mostly been achieved, and that our design for identity constraints is more nearly analogous to that for value constraints than the alternatives described in Section 6. The principal difficulty here is unconstrained aliasing, which does not seem entirely harmonious with the existing mechanism for constraints on values. In Kaleidoscope'93, we have chosen to live with unconstrained aliasing, despite its problems — all of the other solutions we considered either exhibited the same capacity to construct unconstrained aliases, or were insufficiently expressive (see Section 6). Also note that unconstrained aliasing is not new with CIP languages — any language with pointers, and in particular all traditional object-oriented languages, allow unconstrained aliasing.

The question as to whether Goal 5 is achieved (the object identity model should be a natural evolutionary step from current practice) is more subjective; however, again we believe that it is, particularly when the Kaleidoscope semantics is presented in terms of the perturbation model of constraints discussed in Section 4.2. Efficiency (Goal 6) is much improved over our Kaleidoscope'90 implementation, but is still very slow compared to that of a well-engineered implementation of a more conventional object-oriented language. We are continuing to work on improving our implementation — a key issue here will continue to be avoiding costly runtime constraint satisfaction by doing as much constraint satisfaction as possible at compile time.

4.5 Type Invariants

Besides addressing our goals, identity constraints enforce particular categories of invariants on instances of a class, similar to Klarlund and Schwartzbach's graph types [Klarlund & Schwartzbach 93]. For example, doubly-linked lists in Kaleidoscope'93 can be expressed as:

```
class DList subclass of Object;
  var prev: DList;
  var next: DList;
  var cell;
end class DList;

procedure initially (d: DList)
  always: if d.next ≠ nil then d.next.prev == d?;
end procedure initially;
```

The initially procedure is similar to a C++ constructor or Emerald initially clause, and is executed every time an instance of DList is created. This procedure adds a constraint which will cause all operations on DList cells to obey the doubly-linked invariant. A programmer writing a procedure to performing list surgery, such as insert_before or insert_after below, need not explicitly include code to enforce invariants on cells. In contrast, insertion in a strictly imperative language would require operations to make sure that two newly linked cells mutually reference each other, by re-satisfying implicit constraints on elements of linked lists. Since type invariants are explicit in Kaleidoscope'93, program errors are avoided, since methods cannot contradict constraints on members of a type. Type invariants may also include value constraints in addition to identity constraints.

```
procedure insert_before(new: DList, d: DList)
  if d.prev ≠ nil then
     d.prev.next :== new;
  new.next :== d;
end procedure insert_before;

procedure insert_after(new: DList, d: DList)
  var temp: DList;
  temp :== d.next;
  d.next :== new;
  new.next :== temp;
end procedure insert_after;
```

There are particular class invariants that are handled by graph types but that cannot be enforced by identity constraints, such as specifying binary trees where all leaves are linked in a cycle. Identity constraints can be used to define an instance of such a class, but identity constraints cannot maintain this invariant after making arbitrary mutations to an instance. On the other hand, identity constraints differ from graph types since they allow one variable to track another variable for some limited duration of time.

5 Implementation

In the Kaleidoscope'93 VICS constraint framework [Lopez et al. 93], constraints are grouped and solved according to their type: class/type constraints, identity constraints, and value/structure constraints. This independence allows Kaleidoscope to use a different specialized solver for each group, rather than a single more general, but less efficient, algorithm that can handle all kinds of constraints.

In the Kaleidoscope virtual machine, variables are implemented as slots containing pointers to objects, and objects as sets of slots, i.e., a contiguous region of memory.

Identity constraints are implemented as equality constraints on the values of the slots, whereas value constraints constrain the contents of the memory pointed to by the slots. Using the VICS framework, identity constraints are satisfied before satisfying value constraints due to the need to determine the identity of an object (its address in memory) before determining its value (the contents of the memory at that address). The identity-equality constraints are efficiently solved by our current incremental local propagation constraint hierarchy solver, CobaltBlue. (Further information on our local propagation algorithms can be found in [Freeman-Benson et al. 90, Sannella 93].)

Design Goal 6 states that identity constraints should be efficiently implementable. Our constraint solver is quite efficient (see [Sannella 93] for details). However, given the ubiquitous nature of constraints in Kaleidoscope (many of which are trivial), higher performance can be achieved by not using the solver at all. Due to the explicit nature of constraints in Kaleidoscope (as opposed to the implicit nature of aliasing in a traditional object-oriented language), the Kaleidoscope compiler can pre-solve many constraints and thus bypass or reduce the run-time use of the constraint solver. While we have not yet begun to develop this capability in earnest, our current compiler detects and pre-solves several very common uses of identity constraints, including those used in assignments, parameter passing, and stays.

6 Some Plausible Alternative Designs

We considered many other designs for integrating constraints and object identity in Kaleidoscope'93. Each of these designs failed to meet some of the design goals listed in Section 2. We describe here some of the most plausible ones as an illustration of the design space from which we chose the Kaleidoscope'93 solution.

6.1 Object Identity Not Semantically Significant

As a result of having constructs for object identity, traditional object-oriented languages allow unconstrained aliasing. Unconstrained aliasing poses additional problems for CIP languages, since these undeclared relations can yield unexpected results for value constraints. One solution that we considered was simply to eliminate object identity as a part of the language semantics, relegating it to part of the compiler implementation strategy only. In other words, such a design would prohibit constructs that enable variables to bind to a specific object, and hence aliasing. This design was adopted by Kaleidoscope'90. Equality constraints replace object identity in most cases.

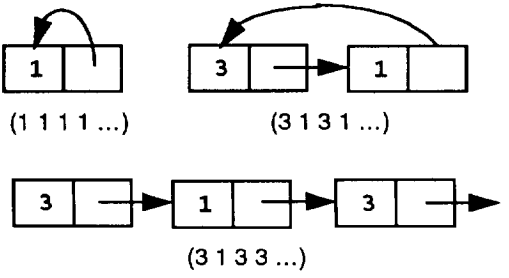
Probably the most important defect of this design is that object identity is fundamental to the use of object-oriented programming for systems modelling, and should hence be part of the language semantics. In addition, there were problems coding particular programming idioms. For example, consider the circular list example in Section 3:

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

Without a way to refer to the identity of an object, this object could be represented by an infinite list with no particular choice of structure. However, if the constraint:

```
a.tail.head = 1;
```

were added, then the choice of object structure to satisfy the initial constraints becomes visible, e.g.,



Without operations to control identity, the programmer has little control over these choices for object structure. Representing structures such as threaded trees, doubly linked lists, and cyclic graphs is difficult without some means of controlling object identity.

6.2 Conventional Object Identity

In a typical object-oriented language, after an assignment statement is executed, the variable on the left hand side of an assignment refers to the object generated from the expression on the right hand side. If the expression on the right hand side is a variable, then the two variables are aliased. For instance, the statement:

```
a := b;
```

causes variables a and b to refer to the same object. Message sends to one of the aliased variables will be visible by the other variable (Figure 1c). Note that assignment produces no explicit constraint between a and b. This model is used in Smalltalk, Common Lisp, and other languages.

The obvious advantage of identity assignment is that aliasing is simple and familiar. This would satisfy our goals of being able to represent objects with self-reference (e.g., circular lists), supporting a straightforward semantics for aliasing two variables, and designing an object identity model that is plausible to the imperative programmer.

Conventional object identity falls short of Goal 4, since conventional identity constructs do not use constraints. If alpha were aliased to beta by assignment, then value constraints on alpha would mutate beta even though there is no constraint between the two variables. This would be a satisfactory solution in a language without constraints, but we would like all (or at least most) relationships between variables to be visible among the active constraints.

Conventional object identity is a special case of the identity constraints described in Section 4. Identity assignment is equivalent to restricting the language to use once

one-way identity constraints. Such once constraints allow the identity of a variable to be changed without the creation of a long-lived constraint to continue enforcing the relationship. Many situations where one would use a conventional identity construct can be replaced by constructs such as always identity constraints, which protect against unconstrained aliasing. Furthermore, always identity facilitates call-by-identity in binding actuals and formals as described in Section 4.1. Generalizing identity assignment in this way is powerful, since it allows identity relationships to persist beyond the initial identity change.

Specifying identity using constraints is an appealing design goal and is well-suited for the CIP family of programming languages. In previous work, we modelled assignment as constraints across successive states of a system. Identity as a constraint continues this theme of modelling imperative constructs with declarative constraints.

6.3 Pointer Variables

The pointer variable solution uses standard equality to achieve the effect of identity. To alias two variables, a and b, an equality constraint is placed between pointer variables referencing a and b as follows:

```
a, b: List;
x, y: ref(List);
x = ref(a);
y = ref(b);
always: x = y;
```

All aliasing is performed by equating these pointer variables. The major advantage of pointer variables is that aliasing is simply normal equality between pointer variables, a familiar construct to CIP programmers. The advantage over the Kaleidoscope'93 design is that there is no distinction between identity constraints and value constraints since ordinary equality constraints are used for aliasing.

One drawback of pointer variables is the extra complexity they introduce — programmers must now distinguish between normal and pointer variables when writing code. Additionally, using the same symbol for two distinct concepts (value equality and object identity) is confusing, much as the heavily overloaded & symbol can be confusing to C++ programmers. We believe that introducing extra complexity for the programmer just to simplify the implementation is a poor language design — the goal of Kaleidoscope is to make reading and writing programs easier!

Similar to identity constraints, the pointer variables alternative satisfies all of the design goals. It is as general as multi-way identity constraints, since equality constraints can be annotated with different durations, or restricted to a one-way constraint by use of read-only annotations. Unconstrained aliasing is also a possible consequence of using pointer variables. This occurs if two variables are aliased for a fixed duration, since they remain aliased by the default *once* equality constraint that is no longer active.

As should be clear from the above discussion, a design using pointer variables would be a very reasonable alternative to the one we selected. Until we gain more experience with writing Kaleidoscope programs, it is difficult to say that one design is clearly preferred over the other.

7 Related Work

Aliasing has long been a concern in programming language design. Languages such as Euclid [Lampson et al. 81] impose syntactic restrictions to eliminate unconstrained aliasing. Khoshafian and Copeland [Khoshafian and Copeland 86] relate the concept of object identity as used in programming languages and in databases. In object-oriented languages, the concept of islands is useful in protecting the user from some of the pitfalls of aliasing while supporting essential uses [Hogg 91]. Identity constraints serve as a form of alias advertisement [Hogg et al. 92], by making identity relationships explicit.

There has also been a great deal of research on detection of aliases in the compiler community [Landi & Ryder 92]. Static detection of aliases facilitates code optimizations for cases where code can guarantee no aliasing. In CIP languages, the ability to bypass expensive constraint operations by compiling imperative code to perform these operations is enhanced by detection of aliasing. Utilizing these techniques will improve performance of generated code for constraint satisfaction.

Graph types are a declarative means of describing invariants on object structure [Klarlund & Schwartzbach 93]. Identity constraints are similar to graph types in their ability to maintain invariants on structure such as preserving doubly linked list invariants while performing list surgery on doubly linked lists. Unlike identity constraints, graph types do not tie one particular variable to another. Instead, objects are linked by navigating through regular expressions over the object structure. If the object structure changes, then these links change, allowing objects to reference different objects in order to satisfy invariants on the data type. Since identity constraints link specific variables, invariants on a data type which depend on particular variables to be linked can be modelled by identity constraints. Identity constraints cannot enforce relations whose identity depends on a general change to the structure of an object.

In CIP languages, Kaleidoscope'90 used equality constraints in place of object identity; object identity was regarded as an implementation technique for equality and was not considered to be a part of the language semantics. In other work on CIP languages, Siri [Horn 92, Horn 93] has a traditional notion of object identity, but restrict constraints to operate within the bounds of their defining object. Li's language [Li 92] provides always and for-now constraints rather than always and once. The active duration of a for-now constraint is until another for-now constraint is enforced on the same variable, i.e., its active duration is the same as its effective duration. We considered for-now durations for identity constraints in Kaleidoscope'93, but for-now does not eliminate or reduce unconstrained aliasing. Rather than include additional durations in the language, we decided to remain with the familiar once, always, and during model.

An different approach to the problems introduced by object identity is simply to outlaw it. This is the approach taken, for example, in pure functional programming languages, such as Miranda or Haskell [Hudak 89]. Similarly, in pure logic programming languages, there is no notion of object identity as such: if variables x and y refer to two ground atoms or terms, it is of no concern to the programmer whether in the implementation there is one shared term or two equal terms. This is also the case for variables, or terms containing variables — unifying two logic programming objects is best regarded as setting up an equality constraint between them; again, whether structure is shared or not should not be visible to the programmer. (Indeed, one can show that pure Prolog is an instance of the Constraint Logic Programming language scheme [Jaffar & Lassez 87], where the domain of the constraints is the Herbrand Universe.)

In concurrent logic programming languages [Shapiro 89] and in the cc (concurrent constraint) languages [Saraswat 89], it is possible to model object-oriented programming, including state, by representing objects as perpetual processes that consume a stream of messages [Kahn 89]. In these languages, each reference to an object is represented as a message stream. To have two different references to an object, it is necessary to have two separate streams, which are then merged to produce a single message stream that is fed to the process representing the object. Recently, Saraswat and others have generalized the cc paradigm to timed concurrent constraint (tcc) programming [Saraswat et al. 94], permitting the programming of reactive real-time systems. These tcc languages are quite close to the CIP language family in spirit. (Some important differences are that the cc and tcc languages support concurrency and allow a store containing partial information, while Kaleidoscope supports constraint hierarchies for expressing defaults and preferences.) However, support for object identity in the cc and tcc languages is handled in effect as in the design described in Section 6.2; object identity is not specified using constraints.

8 Concluding Remarks

While there are a number of plausible designs for integrating declarative constraints with imperative-style object identity, only two meet all six of our goals. We chose the one-way identity constraint design, rather than the pointer variable equality design, because we preferred to keep the Kaleidoscope'93 semantics as simple as possible. Section 4.3 demonstrated that identity constraints do not eliminate unconstrained aliasing, and the example in Section 4.1 illustrated that such aliasing can be intended and useful. In many cases, however, using always identity constraints can reduce unconstrained aliasing as well as assist the compiler in producing better code.

Graph types are capable of satisfying complex constraints on structure that cannot be enforced by value and identity constraints in existing CIP languages. We plan to investigate extensions to identity constraints to enforce a wider set of constraints on object structure. Such an extension would allow complex invariants on object structure to be enforced after an object is initialized, thus simplifying the programming task for such data structures, and ensuring that methods do not violate class invariants.

We have argued that, just as object identity is an essential concept in standard object-oriented languages, it is also essential in CIP languages. In CIP languages, identity constraints can provide a powerful, understandable and efficiently implementable approach to supporting object identity.

Acknowledgments

Many people have given help and advice on this work; we would like to thank in particular Craig Chambers, Denise Draper, Jens Palsberg, Michael Sannella, and Michael Schwartzbach.

This work has been supported in part by the U.S. National Science Foundation under Grant No. IRI-9102938, by the Canadian National Science and Engineering Resource Council under Grant No. OGP-0121431, by a Fellowship from Apple Computer and a Graduate Research Assistantship from the University of Washington Graduate School for Gus Lopez, and by Academic Equipment Grants from Sun Microsystems.

References

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

[Dahl & Nygaard 66] Ole-Johan Dahl and Kristen Nygaard. SIMULA - An ALGOL-Based Simulation Language. *Communications of the ACM*, 9(9): 671-678, September 1966.

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

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

[Freeman-Benson & Borning 92] Bjorn Freeman-Benson and Alan Borning. Integrating Constraints with an Object-Oriented Language. In *Proceedings of the 1992 European Conference on Object-Oriented Programming*, pages 268-286, June 1992.

[Hogg 91] John Hogg. Islands: Aliasing Protection In Object-Oriented Languages. In *Proceedings of the 1991 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Phoenix, Arizona, pages 271-285, October 1991.

[Hogg et al. 92] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva Convention on the Treatment of Object Aliasing. OOPS Messenger, 2(3): 11-16, April 1992.

[Horn 92] Bruce Horn. Constraint Patterns as a Basis for Object-Oriented Constraint Programming. In *Proceedings of the 1992 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, British Columbia, pages 218-233, October 1992.

[Horn 93] Bruce Horn. *Constrained Objects.* PhD Thesis, Carnegie-Mellon University, Computer Science Department, November 1993. Published as Carnegie Mellon School of Computer Science TR CMU-CS-93-154.

[Hudak 89] Paul Hudak. Conception, Evolution, and Application of Functional Programming Languages. *Computing Surveys*, 21(3): 359–411, September 1989.

[Jaffar & Lassez 87] Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *Proceedings of the Fourteenth ACM Principles of Programming Languages Conference*, Munich, pages 111-119, January 1987.

[Kahn 89] Kenneth Kahn. Objects — A Fresh Look. In *Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 207-223, July 1989.

[Khoshafian & Copeland 86] Setrag Khoshafian and George Copeland. Object Identity. In *Proceedings of the 1986 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, pages 406-416, September-October 1986.

[Klarlund & Schwartzbach 93] Nils Klarlund and Michael Schwartzbach. Graph Types. In *Proceedings of the 1993 ACM Principles of Programming Languages Conference*, June 1993.

[Krogdahl & Olsen 86] S. Krogdahl and K.A. Olsen. Modular and Object-Oriented Programming. *DataTid* No. 9, Sept. 1986 (in Norwegian).

[Lampson et al. 81] Butler Lampson, James Horning, Ralph London, James Mitchell, and Gerald Popek. Report on the Programming Language Euclid. Technical Report CSL-81-12, XEROX Palo Alto Research Center, October 1981.

[Landi & Ryder 92] William Landi and Barbara Ryder. A Safe Approximate Algorithm for Interprocedural Pointer Aliasing. In *Proceedings of the 1992 ACM Conference on Programming Language Design and Implementation*, San Francisco, California, pages 235-248, June 1992.

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

[Li 92] Jiarong Li. *Integrating constraints into existing graphical programs.* Unpublished manuscript. Swedish Institute of Computer Science, 1992.

[Lopez et al. 93] Gus Lopez, Bjorn Freeman-Benson, and Alan Borning. Kaleidoscope: A Constraint Imperative Programming Language. University of Washington Computer Science & Engineering Technical Report 93-09-04, September 1993. To appear in *Constraint Programming*, B. Mayoh, R. Tõugu, J.

Penjam (Eds.), NATO Advanced Study Institute Series, Series F: Computer and System Sciences, Springer-Verlag, 1994.

[Madsen et al. 93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[Sannella 93] Michael Sannella. *The SkyBlue Constraint Solver*. Technical Report 92-07-02, Department of Computer Science & Engineering, University of Washington, February 1993.

[Saraswat 89] Vijay Saraswat. *Concurrent Constraint Programming Languages*. PhD Thesis, Carnegie-Mellon University, Computer Science Department, January 1989. A revised version is published as Vijay Saraswat, *Concurrent Constraint Programming*, MIT Press, 1993.

[Saraswat et al. 94] Vijay Saraswat, Radha Jagadeesan, and Vineet Gupta. Timed Concurrent Constraint Programming. To appear in *Constraint Programming*, B. Mayoh, R. Tõugu, J. Penjam (Eds.), NATO Advanced Study Institute Series, Series F: Computer and System Sciences, Springer-Verlag, 1994.

[Shapiro 89] Ehud Shapiro. The Family of Concurrent Logic Programming Languages. *Computing Surveys*, 21(3): 412–510, September 1989.