A Systematic Analysis of Assignment Primitives

Scott M. Pike
Texas A&M University
pike@cs.tamu.edu

Wayne D. Heym Bruce Adcock
Derek Bronish Jason Kirschenbaum
Bruce W. Weide
The Ohio State University
w.heym@ieee.org, {adcockb, bronish, kirschen, weide}@cse.ohio-state.edu

Abstract

Data movement in nearly all modern imperative languages is based on a single primitive: traditional assignment. (With traditional assignment, data are moved between variables by copying.) Unfortunately, traditional assignment poses many known software engineering drawbacks with respect to efficiency for value types, and with respect to modular reasoning for reference types. Moreover, its entrenched legacy has stifled serious regard of potentially superior data-movement primitives. Exploration of the complete design space for data-movement primitives supports the following conclusions: (1) traditional assignment is fundamentally flawed, and (2) *any* other data-movement primitive would be better.

Keywords aliasing, assignment, data movement, parameter passing, swapping

1. The Data Movement Question

The issue of how to achieve data movement between variables is a technical problem that must be addressed by all software developers and by all designers of imperative programming languages. We pose this problem as the **data-movement question**: How does some variable (say, x) get the value of another variable (say, y)?

Before answering this question, many people legitimately ask another: Why would one ever want to make some variable (say, x) get the value of another variable (say, y)? In other words, if the value of y is needed at some point in a program, why not just use y directly, rather than first making x get that value and then using x? Some of the more common and important reasons follow.

1. Parameters passed to (and results returned from) function calls must be transmitted between callers and callees. For example, the value of a formal parameter (say, x) must get the value of a corresponding actual parameter (say, y) when an operation declared as "P(x)" is invoked by a client as "P(y)".
2. Sometimes a value needs to be remembered for future use (*e.g.,* an intermediate result, checkpoint data, etc.) or for separate use in two or more computations.
3. Repetition constructs must establish their loop invariants at the end of each iteration. This often involves making the value of some variable (say, x) — denoting some important quantity at the beginning of the loop body — get the value of a corresponding variable (say, y) at the end of the prior iteration of the loop body.
4. When storing elements into a collection, the code implementing the collection component must make some variable (say, x) in a collection's representation get the value of another variable (say, y) that is to be stored there.

The canonical answer to the data-movement question — implicit in case 1 and explicit in cases 2, 3, and 4 — is to use the traditional assignment primitive "x := y" (also written as "x = y" in many languages). This built-in solution to data movement runs thick in the bloodline of all popular imperative languages, *e.g.,* Fortran, C, C++, Java, and C#.

Operationally, traditional assignment is straightforward. At the level of hardware instruction sets, traditional assignment simply copies the data stored in location y into location x, replacing the data previously stored in location x. This is something that all computer hardware is designed to do efficiently, so it is no surprise to see a direct high-level-language manifestation of such an important low-level mechanism. Virtually all modern software systems have been engineered with this data-movement primitive in mind. The choice of traditional assignment as *the* built-in primitive for data movement seems so basic, obvious, and authorized as not to merit serious reflection or reconsideration.

We claim that this received viewpoint is unwarranted; the traditional assignment primitive is fundamentally flawed and leads to suboptimal software designs in several critical dimensions. The goal of this paper is to initiate a departure from the past — one aimed at breaking the genealogical grasp of traditional assignment on both software design and programming language design.

We start by revisiting some known problems with traditional assignment (also called **preserving assignment** hereafter) in Section 2. In Sections 3 and 4, we characterize and explore the full solution space of alternative answers to the data-movement question. Our analysis in Section 5 concludes that **exchanging assignment** (swapping), **replacing assignment**, and **destructive assignment** are all superior to traditional preserving assignment as a foundation for software engineering and programming language design. Fundamentally, these alternatives better support modular reasoning about software system behavior and lead to improved efficiency, simpler storage management, and easier software component specification and understanding.

The contributions of this paper are in recognizing, characterizing, and analyzing a surprisingly simple question that is, in essence, a cornerstone of software design. We develop a common analytical framework for integrating and unifying prior work on data movement. Furthermore, we suggest how relatively simple changes in imperative languages can enable software engineers to take advantage of non-traditional data-movement primitives.

2. Technical Framework

This section outlines technical and historical factors that underpin the analysis in Sections 3 and 4. We postpone a broader treatment of related work to Section 6, in which we survey the impact of various assignment primitives on programming languages and modular reasoning systems.

2.1 Different Kinds of Right-Hand-Sides

We begin by noting that traditional assignment, despite some languages' typographical evidence to the contrary, is an asymmetric operation. For example, in a language where assignment is written "x = y", x and y play different roles. The recipient of data is x, and the source of the data is y.

There seems to be no room for debate over how an assignment of y to x impacts x: it gets the value indicated by y. There are also cases where the impact of such an assignment on y seems incontrovertible. For example, if y is a declared constant or a literal, it is unchanged by the assignment. Furthermore, if y is an expression or a call to a side-effect free function, the universally understood meaning is first to evaluate y and then to let x get that result. We therefore proceed in the rest of the paper to discuss only how an assignment of y to x impacts y in the problematic case: when y is just a "bare" variable.

It should be noted at the outset that a move away from traditional assignment does not mean a total abandonment of aliasing, nor of value-copying. To achieve the effect of traditional assignment (*e.g.,* x := y) in the presence of a new assignment primitive (say, ←), one can introduce a function that returns a copy of whatever is passed to it, be it reference or value. Replica seems to be a good choice for this function's name. By the previous discussion, one can see that there is only one reasonable meaning for x ← Replica(y), namely, set the value of x to be the value returned by Replica(y). Therefore, the effect of traditional assignment for *both* values and references can still be achieved in the cases where it is desired, even when using non-traditional assignment operators.

2.2 Aliasing and Reasoning Complexity

A well-known folk theorem in computing circles is that "all problems in computer science can be solved by another level of indirection, but that will usually create another problem."[1] Like most folklore, this claim is partially true — a fact not lost on programming language designers, who have consistently delivered a variety of language constructs to make it easier to write programs that use indirection. Unfortunately, one problem not solved, but rather exacerbated, by indirection is reasoning about software system behavior. The result of using indirection is the capacity for aliasing, which is widely acknowledged as a primary source of reasoning difficulty for imperative programs.[2]

Indeed, it is not just actual aliasing, but the mere *potential* for aliasing that complicates both formal specification and verification, and hence informal understanding of — and informal (but sound) reasoning about — software system behavior [Weide and Heym 2001]. This has been widely known for decades. As early as 1973, Hoare remarked of pointers that "their introduction into high-level languages has been a step backward from which we may never recover" [Hoare 1989]. In 1976, Kieburtz explained why we should be "programming without pointer variables" [Kieburtz 1976]. Cook's seminal 1978 paper on the soundness and relative completeness of Hoare logic identified aliasing (of arguments to calls, *i.e.,* even in a language without pointers or reference variables) as the key technical impediment to sound modular verification of imperative programs [Cook]. In writing about aliasing, Hogg noted that, "The big lie of object-oriented programming is that objects provide encapsulation" [Hogg 1991], which is not the case when aliasing can cross putative encapsulation boundaries.

2.3 The Traditional Assignment Paradigm

As a data-movement primitive, traditional assignment exacerbates the conflict between the inefficiency of value types and the (aliasing-induced) reasoning problems of reference types. Traditional assignment has two related interpretations in modern high-level imperative languages, depending on whether x and y are variables of a value type or variables of a reference type. If x and y are variables of a value type, *e.g.,* int, then copying leaves x and y independent of each other, so all subsequent changes to x do not affect y, and vice versa. This is fine for built-in types, which are small, but it

[1] This aphorism, which is frequently attributed to Turing Award winner Butler Lampson, is actually due to David Wheeler, inventor of the subroutine.

[2] By *aliasing*, we mean what is often called *visible aliasing*; that is, having multiple names for a *mutable* entity.

becomes unacceptably inefficient for user-defined types with potentially mammoth representations.

On the other hand, if x and y are variables of a reference type, then the references x and y subsequently can be changed independently, *i.e.,* without affecting each other. But the objects referenced by x and y are *not* independent, because subsequent method calls that change the object referenced by x also change the object referenced by y, and vice versa. This difference between copying values and copying references is evident in the two common parameter-passing mechanisms of call-by-value and call-by-reference.

In the case of reference types, an additional implication is that traditional assignment creates aliases, which (if observable) dramatically complicate the formal specification [Weide and Heym 2001] and modular reasoning about program behavior [Hogg et al. 1992]. Note also that when reference types are involved, after a traditional assignment statement there is generally one more reference to some object and one fewer reference to another object, so there are storage management implications either for (1) the programmer in a language such as C++, or for (2) the garbage collection mechanism in languages such as Java and C#.

The question of the "observability" of aliasing arises because an object that is referenced might be either mutable (capable of having its value changed) or immutable (having a fixed value upon creation). No reasoning problem arises from aliases to an immutable object, because that object's value can't be changed through any such reference; there might as well be just one copy of the object for each reference. In other words, a reference type where the referent is immutable can be thought of as a value type. There may or may not be performance and storage management differences associated with the distinction (depending on the referent type), but there are no reasoning differences.

Despite the potential for reasoning problems, both traditional assignment and the value-reference type dichotomy have been codified into modern commercial software technologies, including C++, Java, and the .NET framework. From a software engineering standpoint, this is a step backwards into ontological dualism. That is, programmers must be aware that variables of some types have ordinary values, while variables of other types hold references to objects (where the objects have the *actual* values). Such dualism undermines a simple, coherent, and unified view of specifying and reasoning about types. Ideally, we would like a uniform type ontology that provides the reasoning simplicity of value types together with the execution efficiency of reference types. Simply put, the traditional assignment primitive (as a solution to the data-movement problem) falls short of this goal.

For parameterized components — which have become popularized by generic programming constructs such as templates — this creates a special problem. Inside a component that is parameterized by a type Item, there is no way to know prior to template instantiation time whether traditional assignment of one Item to another will copy a value or a reference. Of course, we can hack a "fix" as it is done in Java by introducing otherwise-redundant reference types to immutable objects (such as Integer) that correspond to value types (such as int). Actual template parameters can then be limited to reference types, but the bloated type ontology exhibits a marked lack of parsimony. The technique known in .NET (and now Java) as "boxing" is a cosmetic improvement in terms of syntax, but fails to remove the value-reference dichotomy or the need to understand its ramifications for program behavior.

Alternatively, we can adopt a design discipline that tries to make user-defined class types act like value types rather than reference types. A popular approach in this direction is Coplien's orthodox canonical form for C++ [Coplien], in which assignment and copy constructors make "deep copies", so that "variables created from your classes can be assigned, declared, and passed as arguments just like any C variable" (page 38). Indeed, users of the C++ Standard Template Library (STL) are told to follow this advice [Musser et al. 2001]. Accordingly, we refer to this traditional assignment paradigm as **preserving assignment**, since it preserves the value of the right-hand side parameter.

In summary, going the direction of making everything a reference rather than a value only exacerbates the complications for specification and modular reasoning that are caused by potentially aliased references. It is true that the reasoning problems created by aliasing could, in principle, be avoided by requiring that *all* reference types should refer to immutable objects. This would be tantamount to mandating pure functional programming in a nominally imperative language, and to our knowledge no one is seriously proposing this as the light at the end of the tunnel on the moving-toward-references track.

Indeed, the prevailing advice for software component designers and clients would be better characterized as moving-toward-values — except that while this works in C++, it does not work in other popular languages that provide little or no flexibility for overriding assignment and copy constructors (*e.g.,* Java). Hence, the collection classes in java.util, for example, have been designed with preserving assignment of references as the underlying data-movement primitive; and there is little a client using these classes can do about it.

2.4 Exchanging Assignment

One alternative data-movement primitive, **exchanging assignment** or swapping [Kieburtz 1976, Harms and Weide 1991], has been studied more carefully and used more systematically than the other two equivalence classes of alternatives evaluated in Section 3. Like preserving assignment, exchanging assignment has been used as the basis for the design of a research language, Resolve [Sitaraman and Weide 1994], and associated component libraries. Resolve, adapted

as a discipline for C++ programming, has been evaluated in both educational and industrial software settings, including a 100K+ SLOC commercial software product [Hollingsworth et al. 2000]; the code shown later in this section is representative of what is in that product. Previous work [Harms and Weide 1991, Hollingsworth et al. 2000] also has addressed how exchanging assignment interacts with other programming language issues (*e.g.,* parameter passing) and software design issues (*e.g.,* application-induced requirements for deep copies). With exchanging assignment, the answer to the data-movement question becomes: "x :=: y". For instance, if x = -42 and y = 97 before x :=: y, then x = 97 and y = -42 afterward.

We use exchanging assignment to illustrate how both software component design and client code differ when preserving assignment is not taken for granted as the underlying data-movement primitive. As an example, consider a FIFO queue component with methods to Enqueue and Dequeue an item from the queue, to get the queue's Length, and perhaps others. Most of these methods do what anyone using a FIFO queue component might expect from experience with a component design based on traditional assignment. However, Enqueue is different: it *moves* the argument value onto the queue and hence changes the argument's value in the client program. In the component design discipline used in the study mentioned above [Hollingsworth et al. 2000], an appropriate value for the argument upon return from Enqueue is an initial value for the argument's type; but this is not the only design possible.

How is such a queue component used in a client program? Here is sample client code for finding and removing a smallest item from a non-empty queue q, returning it in min, and possibly permuting q in the process. This code illustrates one of the few new programming idioms that arise when using exchanging assignment, *i.e.,* how to iterate over a collection by using a "catalyst" variable rather than an iterator [Weide et al. 1994]. (For efficiency, one probably would see an iterator in the STL or java.util version of such code, because these libraries are based on preserving assignment.) The C++ operator &= is used in this code for :=: because there is no native exchanging assignment in C++ . The component designer provides this exchanging assignment data-movement operator, by convention, for each new user-defined type.

```
void Remove_Min (Queue_Of_Item& q, Item& min)
{
   Queue_Of_Item q1;
   q.Dequeue (min);
   while (q.Length () > 0) {
      Item x;
      q.Dequeue (x);
      if (x < min) {
         x &= min;
      }
      q1.Enqueue (x);
   }
   q &= q1;
}
```

Perhaps surprisingly, as illustrated here, using exchanging assignment demands few changes in programming style [Hollingsworth et al. 2000]. One's investment in software engineering and programming knowledge is not lost merely because the data-movement primitive is different.

As we will explain further below, exchanging assignment is safe with respect to modular reasoning, because the effect of data movement does not introduce aliasing. Moreover, it allows all types to be viewed as values *without compromising efficient execution*. Types with small representations can be exchanged directly by swapping values. The component designer (or, if :=: is a language primitive, the compiler) can introduce one level of *hidden* indirection for representations larger than a given threshold, and swap pointers to the representations of x and y in order to effect the *logical exchange* of x and y as values. To a large extent, the other two alternatives to preserving assignment also offer these relative advantages with comparably subtle — but crucial — changes in component design and client coding idioms, as we see next.

3. Solution Space and Criteria

First, we need to characterize the possible ways one could define a (binary) data movement primitive. We use "←" to denote this primitive operator — in English we might pronounce it "gets" — so by definition the answer to the data movement question is: x ← y. Again note that y here is simply a "bare" variable, since the cases of assigning the result of an expression evaluation or a function call are generally unproblematic with respect to data movement. The key to identifying and classifying all possible meanings for ← is to note that while ← is required to leave x with the old value of y in order to solve the stated problem, there is complete flexibility in the way that ← provides a new value for y. Pragmatically, these alternatives can be subsumed under four equivalence classes that characterize the possible values of y after executing x ← y:

1. **Destructive Assignment:** y has no value (undefined)
2. **Replacing Assignment:** y has a legal value of its type, chosen from a statically-specified set of values

3. **Preserving Assignment:** y has its own old value
4. **Exchanging Assignment:** y has x's old value

These exhaust the possibilities that make any sense. There are only two *specific* values in sight as the problem is stated: the old value of y and the old value of x. Other than these, we can select the new value of y without regard to the old values of x and y, from among all the values of its type, in two ways: we can give y no value (the "undefined" option), or we can give it some value chosen arbitrarily (non-deterministically) from among a set of admissible values. This second case could be further partitioned in obvious ways; for example, considering only a singleton set of admissible values determines a unique statically-specified final value for y. It turns out that nothing hinges on this further partitioning. Also, non-deterministic choice could also be replaced by probabilistic choice of some kind, but again it turns out that this does not affect any of our conclusions. Hence we do not further discuss these variants.

The four possibilities above apply both for values and references. First, suppose the type of x and y is int, a value type. If beforehand, x = -42 and y = 97, then after x ← y we must have x = 97. But we could leave y undefined (*i.e.,* unusable in subsequent computations until it gets reinitialized with a new value of type int); we could leave y with some statically-specified value (*e.g.,* a natural choice for type int might be 0); we could leave y with its old value (*i.e.,* 97); or we could leave y with x's old value (*i.e.,* -42).

Now suppose the type of x and y is a reference to objects of type T. If beforehand, x = ref-to-B and y = ref-to-A (where A and B are objects of type T), then after x ← y we must have x = ref-to-A. But we could leave y undefined (*i.e.,* unusable in subsequent computations until it gets reinitialized with a new reference to type T); we could leave y with some admissible value (*e.g.,* a natural choice for all reference types might be null, or possibly ref-to-O, where O is any object of type T); we could leave y with its old value (*i.e.,* ref-to-A); or we could leave y with x's old value (*i.e.,* ref-to-B).

Before analyzing the equivalence classes for data movement operators, we need a set of criteria that might bear on an evaluation of "better" versus "worse". The most important criterion is that the solution should not break modular reasoning about software behavior, because this foundation is required for any scalable software engineering discipline [Weide et al. 1995]. In Section 4, we consider the following desiderata that are obviously of general interest and that could differ from one data movement operator to another.

- **Efficiency**: How much time does ← take to execute? We consider faster to be better.
- **Ease of storage management**: How much complication does ← introduce into storage management? We consider less complication to be better, not only because it is easier to understand, but also because it is likely to mean better efficiency in terms of overall execution time.
- **Ease of specification and reasoning**: How much complication does ← add to the understanding of component specifications and their use in modular reasoning by component clients? Any scalable engineering discipline must be able to verify component properties in isolation without having them "break" when the component is subject to composition. We consider less complication to be better.

This set of evaluation criteria is not necessarily complete. For example, another criterion might be maximization of client knowledge; that is, how much information does the client know about the state of the program after "x ← y" executes? Arguably, more information is better, in the sense that (if all other things are equal) a more-deterministic program is easier to reason about than a less-deterministic one — and in some cases it can be more efficient, too, if clients can use the additional information to their advantage.

Another legitimate (although less general) criterion might be appropriateness for a particular software development paradigm — for example, compatibility with hierarchical subtyping as witnessed by object-oriented development. We suspend such considerations for the moment, but return to them in Section 5.

4. Analysis and Results

Tables 1 and 2 summarize the analysis of this section using a "Good" versus "Deficient" scale for each evaluation criterion from Section 3. Table 1 lists the results for data movement between value types; Table 2 lists the results for data movement between reference types. Each table row corresponds to one of the four alternatives for the value of y after executing "x ← y".

Looking across the rows, we find deficiencies in all but Case #4 (*i.e.,* exchanging assignment). Notably, the row with the most difficulties is Case #3, in which y's new value equals its old value (*i.e.,* traditional preserving assignment). The other two options also could be considered better approaches than preserving assignment, which — as an unfortunate legacy of early languages without user-defined types — has been hardwired into every widely-used commercial software technology.

Next we consider the most significant table entries for each of the possible solutions to the data movement problem. Figures 1 through 4 illustrate the data movement operators and remark on their deficiencies. For Figures 1(a) through 4(a), x and y are int objects. For Figures 1(b) through 4(b), x and y are references to Set-Of-Int objects.

Destructive assignment (Figure 1) has a major disadvantage in that it complicates specification and reasoning about software behavior. Each value and reference type must be augmented with a special value: "undefined."

Case	New value of y after "x ← y;"	Efficiency of ←	Ease of storage management	Ease of specification and reasoning
1	Destructive Assign	Good	Good	***Deficient***
2	Replacing Assign	***Deficient***	Good	***Deficient***
3	Preserving Assign	***Deficient***	Good	Good
4	Exchanging Assign	Good	Good	Good

Table 1. Evaluation summary for data movement operators with respect to value types. Since aliasing is not at issue, support for modular reasoning and ease of storage management are uncomplicated. The primary deficiencies arise for efficiency and ease of specification, principally because potentially large representations are expensive to create (Case #2) and to copy (Case #3), and because specifications and/or their understandability are complicated by undefined and/or under-defined values. See Figures 1–4 for details.

Case	New value of y after "x ← y;"	Efficiency of ←	Ease of storage management	Ease of specification and reasoning
1	Destructive Assign	Good	Good	***Deficient***
2	Replacing Assign	Good	Good	***Deficient***
3	Preserving Assign	Good	***Deficient***	***Deficient***
4	Exchanging Assign	Good	Good	Good

Table 2. Evaluation summary for data movement operators with respect to reference types. The use of indirection makes all four alternatives potentially efficient. As with the value types, however, reasoning is complicated by undefined and/or under-defined and/or null references. Additionally, the potential for aliasing undermines modular reasoning and complicates storage management in the case of traditional preserving assignment (Case #3). See Figures 1–4 for details.

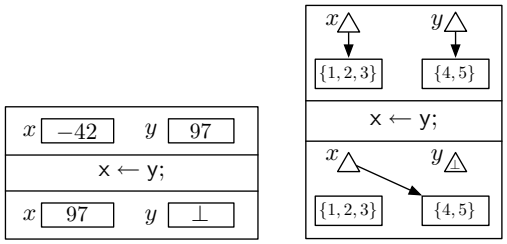
Java programmers, especially those familiar with JML, know the minor mess caused by allowing null values for reference variables. Suddenly, preconditions start looking like: "p $\neq$ null and ..." and/or postconditions start looking like "if #p = null then ... else ..." The same style arises when variables are allowed to be undefined. It is possible to make the problem disappear syntactically simply by making it an implicit proof obligation at each point in the program that, say, none of the variables involved in an expression or a call is undefined (except possibly x in a situation like "x ← ...;"). But it remains incumbent upon the programmer to add this to his/her informal reasoning about program behavior; it's not merely a formality needed in program proofs.

In Figure 2 we see an illustration of replacing assignment. The efficiency concern arises from the cost of constructing and reclaiming values with large representations. For example, the admissible replacing value for a PixelMap value type might be a 2D array of 90,000 integers all initialized to 0. Such costs could be mitigated at the expense of complicating the language implementation with lazy initialization/finalization features. Note also that, while null is an obvious choice for the resulting value of assignment using reference types, it presents the same reasoning problems as in the previous case. The inventor of null references, C.A.R. Hoare, recently went so far as to call them a "billion dollar mistake" [Hoare 2009]. One could instead, as Figure 2(b) shows, use a non-null reference to a new admissible value, but again we return to the efficiency issue of initializing this value.

By far the most prevalent assignment primitive in modern programming languages is preserving assignment, the behavior or which is shown in Figure 3. This primitive suffers from both efficiency and modular reasoning drawbacks. The efficiency problem arises when copying value types with possibly large data representations, which users of the C++ STL are advised to do [Musser et al. 2001]. Reasoning problems arise when traditional assignment is used to copy references/pointers as so-called "shallow copies". Other papers (*e.g.,* [Harms and Weide 1991], [Hogg et al. 1992], [Weide et al. 1995]) have already examined the many problems this causes. Indeed, the traditional assignment operator is the only data movement approach where support for modular reasoning is an issue, because it alone introduces aliased references on its own. All the other solutions are alias-free unless the programmer explicitly wants to create an alias, as explained in Section 2.1, which also simplifies their storage management requirements compared to traditional preserving assignment.

Exchanging assignment is shown in Figure 4, and winds up being the optimal choice in our analysis. This is due to the fact that it incurs neither aliasing nor efficiency penalties. As mentioned earlier, an exchanging assignment can be implemented efficiently to execute in constant time by (i) copy-exchanging small representations directly and by (ii)

introducing a hidden level of indirection for larger representations. The key insight behind why this is possible is that — from the client's perspective — there is no logical difference between exchanging two object references and exchanging the two values of the referenced objects. That is, exchanging the name bindings is logically indistinguishable from exchanging the actual object values.



(a) Destructive assignment for value types. Variable y is *undefined* after "x ← y"

(b) Destructive assignment for reference types. Reference y is *undefined* after "x ← y"

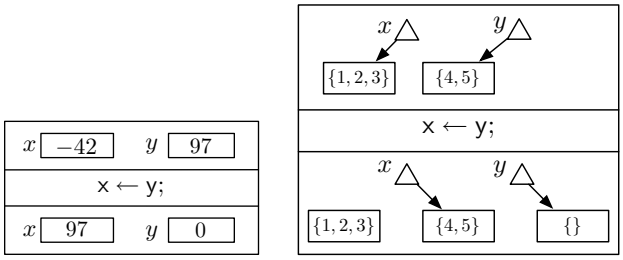
Figure 1. Leaving y undefined requires the introduction of a new value for each type: undefined. This adds additional complication to the programmer's reasoning process; now implementers must specify and clients must understand exactly what behavior modules will have in the presence of this special value. Specification and reasoning are easier if every variable always has a legal value of its nominal type, *i.e.,* if it is always "defined", even if its value at some point is arbitrary or not uniquely specified.

5. Discussion

A careful reader may wonder whether the criteria were "rigged" to produce this outcome. Doesn't exchanging assignment have any problems? An early assessment suggested that programmers who are used to traditional assignment might have to learn a new paradigm of programming in order to use exchanging assignment [Hogg et al. 1992]. However, subsequent experience with using this data movement operator (both in the classroom and in building commercial software) suggests that not much changes for the programmer except the quality of the resulting software [Hollingsworth et al. 2000].

As was mentioned in Section 2.1, it is possible to achieve traditional assignment effects using the other assignment operators. More specifically, the usefulness of indirection mentioned in Section 2.2 remains true to some extent, and programmers will inevitably wish to introduce aliases in some cases. Use of the Replica function discussed in Section 2.1 (*e.g.,* "x ← Replica(y)") is one elegant way to retain this ability regardless of what assignment primitive ← actually denotes, because it makes the right hand side an unproblematic function call.

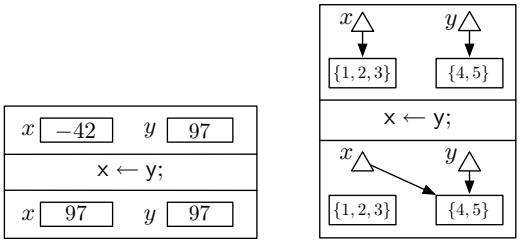
One other potential problem with exchanging assignment has been suggested by Minsky [Minsky 1996]: because of its symmetry, exchanging assignment does not mesh well with the inherently asymmetric notion of subtyping in object-oriented languages. Suppose y is of type S and x is of type



(a) Replacing assignment for value types. Variable y has some (statically) specified value after "x ← y". This figure suggests 0 as a suitable possible value for int objects, but any set of values could be specified.

(b) Replacing assignment for reference types. Reference y has some (statically) specified value after "x ← y". This figure suggests Ref-to-EmptySet as a suitable possible value.

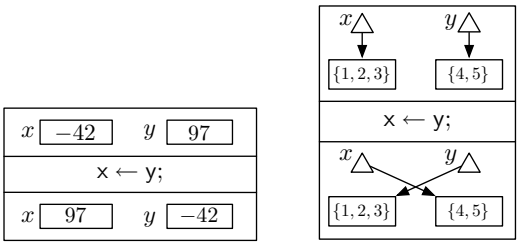
Figure 2. Replacing assignment can be achieved in two different ways. One can either specify a unique statically-determined value for y, or let y take on some value from a statically-determined set of admissible values. Leaving y with a unique specified value of its type is a reasonably good approach, except for the efficiency of ← for value types, particularly those with large representations. Not only must that admissible value be constructed for y, but also the old value of x must be reclaimed. For reference types there is no efficiency problem because null is a perfectly reasonable default value, but of course this introduces the same reasoning issues that afflict destructive assignment. Alternatively, leaving y with one of an admissible set of values is also a reasonable approach. It doesn't suffer from the efficiency concerns because one correct implementation is to specify the set of alternatives as containing all (legal) values of the type, and then simply exchange the values of y and x. The only question, then, is why you wouldn't *want* to tell the client the new value of y in order to maximize the client's knowledge of the values of the program's variables. Not doing so compromises ease of reasoning.



(a) Traditional preserving assignment for value types. Variable y has its original value after "x ← y".

(b) Traditional preserving assignment for reference types. Reference y refers to its original object after "x ← y".

Figure 3. For value types, preserving assignment presents a serious efficiency concern when the value to be preserved has a large representation. Furthermore, for reference types, preserving assignment is undesirable due to the automatic aliasing it induces.



(a) Exchanging assignment for value types. Variable y has the original (old) value of x after "x ← y".

(b) Exchanging assignment for reference types. Reference y refers to the object previously referred to by x.

Figure 4. Exchanging assignment swaps its operands (be they values or references), so no aliases are created to potentially thwart modular reasoning, and no garbage is created to complicate either implicit or explicit storage management. Exchanging assignment also maximizes the client's knowledge of the program state after execution, because it specifies a unique (dynamically determined) value for y which sometimes can be used to advantage in subsequent code.

T, where S is a behavioral subtype of T [Liskov and Wing 1994, Leavens and Weihl 1990]. (The actual objection to exchanging assignment on the grounds of its interaction with subtyping fails to note that if we are not talking about *behavioral* subtypes, then not even traditional assignment should be allowed.) The claimed problem is that O-O programmers want to be able to assign y to x in this case, but not vice versa — so exchanging assignment cannot be used.

We do not give a complete analysis of this issue here, but briefly explain one aspect of it. Indeed, exchanging assignment cannot be permitted where the variables' types do not match, if the situation involves an *explicit* use of exchanging assignment. However, the first and by far most common use of data movement is for parameter passing, and here exchanging assignment can be used even in the presence of subtyping. The declared type of the actual parameter (*i.e.,* in the example, S) must be a subtype of the declared type (*i.e.,* T) of the formal parameter. For purposes of exchanging assignment to pass this parameter to this call to the method M (and again for the return), simply consider the type of the formal parameter to be the same as the type of the actual parameter (*i.e.,* S). Now the actual and formal are swappable. This does not affect the soundness of the separate reasoning about the correctness of M's method body because that reasoning uses T as the type of the formal, and by assumption S is a behavioral subtype of T. It just happens that only T methods happen to be invoked in the body of M.

Even if incompatibility with subtyping were treated as a significant objection to exchanging assignment, either destructive or replacing assignment would still be better than traditional assignment, and they are not subject to the symmetry objection. These two are the best choices for those who prefer asymmetry in their data movement operator in the presence of subtyping.

6. Related Work

Not only is there nothing sacred about traditional assignment as an answer to the data movement question, it is the worst choice among a landscape of better alternatives. We include this brief section on related work to indicate the spectrum of research on problems associated with traditional assignment and aliasing. The primary purpose of this section is to organize alternative operators for data movement that enforce alias *prevention* by virtue of their semantic definition, but without sacrificing efficiency.

6.1 Specification and Verification in the Presence of References and Aliasing

In modern languages, the common store of entities to which references refer is not manifest in the language syntax. As many researchers (as well as Fortran programmers, albeit in the opposite direction) noted long ago, the common store is like an array of entities and the references into it are like indices into that array. Therefore, in order to specify and reason about programs with references, it suffices *technically* to make the common store explicit in the code (as in Fortran programs that simulate pointers with arrays and indices). Euclid, for example, made a common store called a collection an additional parameter to any call with a parameter that was a reference into it. Reasoning about a collection was then just like reasoning about arrays [Horning 1978]. This, or something essentially identical to it, remains the basic approach to specifying and reasoning with references [Weide and Heym 2001].

In the 1970s, specification and verification of programs with pointers was done using the precursors to object-oriented languages, notably Pascal. In the culmination of this line of work, Luckham and Suzuki explicitly modeled the state of memory in specifications and in verification conditions [Luckham and Suzuki 1979]. An important missing ingredient in this early work was any use of abstraction in explaining the behavior of new types, apparently because Pascal lacked user-defined types with hidden representations. In 1980, Ernst and Ogden considered similar specification and verification issues in Modula, which had a module construct with hidden exported types [Ernst and Ogden 1980]. Later, the same authors published a verification method for "shared realizations" of ADTs, including heap storage (*i.e.,* the common store) [Ernst et al. 1994]. This led to a value-based specification, with reference details arising only within the proof of the module implementation. This approach worked under the assumption that the only source of possible aliasing in the language was within that implementation, *i.e.,* not from external client assignment of references.

More recently, related work has involved specification and verification of standard object-oriented software. The problems addressed include specifying behavior of components involving references, and potential aliasing both from copying references and from parameter passing anoma-

lies [Leino and Nelson 2002]. JML [Cheon and Leavens 2002], for example, supports specifications with references and with values.

In summary, the fundamental problem of how to specify and verify programs with reference types is technically solvable by making sure that abstract state variables associated with references "follow them around" throughout specifications, programs, and proofs. However, because the potential for aliasing complicates specification and reasoning, many researchers have sought to limit aliasing.

6.2 Techniques to Limit Aliasing

In the 1990s, this research area experienced a resurgence of activity. Early in this resurgence, Hogg *et al.* provided a classification taxonomy for the area, defining terms for alias detection, advertisement, control, and prevention [Hogg et al. 1992]. Detection is the static or dynamic diagnosis of potential or actual aliasing. Advertisements are annotations that help modularize detection by declaring aliasing properties. Control is the provision of tools and techniques enabling the programmer to "manage" aliasing. Prevention is the use of constructs that rule out aliasing in a statically checkable fashion. Among all these techniques, prevention would seem to afford the greatest simplification of specification and reasoning. It also seems that many of the control techniques might provide greater simplification than would techniques of detection and advertisement. Hogg *et al.*urged that "aliasing must be detected when it occurs, advertised when it is possible, prevented where it is not wanted, and controlled where it is needed" [Hogg et al. 1992]. Due to space constraints here, we list only a few works representing the categories in this classification. A paper summarizing work in all these categories appeared in 1999 [Noble et al. 1999].

The task of creating compiler-generated optimizations has motivated a large body of work on pointer analysis. An extensive summary of this work [Hind 2001] also includes a concise description: "A pointer analysis attempts to statically determine the possible runtime values of a pointer. As such an analysis is, in general, undecidable, a large collection of approximation algorithms have been published that provide a trade-off between the efficiency of the analysis and the precision of the computed solution." Put otherwise, static alias detection may be able to help programmers, but the lack of precision remains a general concern. Programmer-supplied advertisements therefore have been suggested to improve precision [Hendren and Gao 1993].

Work on alias control appears to have begun when Reynolds proposed a method for checking syntactically that pairs of statements obviously do not interfere with each other [Reynolds 1978]. His idea was "to prohibit interference between identifiers, but to permit interference among components of collections named by single identifiers". New approaches to controlling aliases continue to be introduced. Banerjee has shown how standard semantic techniques can be used to assess and compare confinement disciplines proposed in the literature [Banerjee and Naumann 2002].

There is a spectrum of approaches within the prevention category, ranging from enabling prevention to insisting and relying on it. The essential ideas involve mutability and uniqueness of reference. As observed in concurrent-program design [Noble et al. 2000], visible aliasing does not occur when there is only one reference to a mutable object, or when there are multiple references to an immutable object. In his "islands" proposal, Hogg provided an incremental solution working toward encapsulation of the use of aliasing [Hogg 1991]. Minsky also saw the problem of aliasing as being "caused by the almost universal practice in programming to transfer information by copy" [Minsky 1996]. His solution, like Hogg's, employed a destructive read operation. Similarly, alias prevention is enabled by "linear types" [Baker 1995]. Fähndrich has proposed a type system that reduces the usual restrictions on linear types [Fähndrich and DeLine 2002]. Work at the end of the spectrum that insists and relies on alias prevention has been discussed earlier in this paper [Kieburtz 1976, Harms and Weide 1991].

6.3 Practical Impact of Data Movement on Reasoning

These data movement considerations have been applied in several settings. We discuss the impact of the use of data movement operators in the literature in this section.

The language Tako [Kulczycki and Vasudeo 2006] is (essentially) Java with alias avoidance techniques. While this design goal manifests itself in many ways throughout the language, its treatment of assignment primitives is most important here.

Tako attempts to use exchanging assignment whenever possible. Specifically, exchanging assignment is used when the types of the two parameters are exactly the same. The other legal kind of assignment in Tako occurs between objects of two different types, where one is a behavioral subtype of the other. As Section 5 notes, the symmetry of exchanging assignment precludes its use in such situations. Instead, Tako uses replacing assignment in these cases. All types in Tako have an initial value, so the replacing value is chosen to be an initial value for the type of the right-hand side parameter.

The Jahob tool [Zee et al. 2008] is able to verify Java ADT implementations given code annotations for loop invariants, abstraction relations, and representation invariants. While the tool is able to prove many implementations correct and its proofs are sound, the utility to clients of components verified using this approach is still unclear.

For example, an implementation of a java.util.Map-like component as a linked-list data structure was verified by this tool. A Map component provides methods to define new key/value pairs in the map, to remove a particular key/value pair from the map given the key, and to access the value corresponding to a given key in the map. The verified component requires that any object key lookups are

performed using *reference* equality, rather than *value* equality. The java.util.Map interface specification from the java documentation [Sun 2009] instead specifies that key lookups must be performed using the equals() method; keys must be compared as values, not references. The specification of the verified component avoids possible thorny issues relating to preserving assignment for reference types, namely aliasing. This example shows the analysis provided in Section 2.2 is backed by experience; the use of the preserving assignment for data movement complicates reasoning about programs.

As discussed in Section 2.2, Resolve [Sitaraman and Weide 1994] has explored the consequences of using exchanging assignment as the primary data movement operator, with the most recent work [Sitaraman 2009, Kirschenbaum et al. 2008] focusing on automated modular verification of the use of abstract data types. Based on the literature, the analysis in Section 2.2 is accurate; reasoning about programs modularly is possible while maintaining efficiency.

The other possible data movement operators have, to our knowledge, not been implemented by any programming languages and then tested seriously on commercial software projects.

7. Conclusions

Since traditional assignment was introduced at a time when languages had only value types with small representations, it was a perfectly good data movement solution for its day. With the advent of languages having user-defined types and reference types, though, it has become sub-optimal for general use; *i.e.,* it should not be *the* built-in data movement primitive in modern languages. But traditional assignment is deeply woven into the fabric of computing for most software engineers, whose early computing education typically involved programming only with built-in value types having small representations. Languages could still allow traditional assignment in such situations.

The other three data movement primitives all have several advantages over traditional preserving assignment as the built-in data movement primitive that is available for *every* type. Most importantly, they do not interfere with modular reasoning. Some are efficiently implementable for all value and reference types, regardless of the sizes of their data representations. All dramatically simplify storage management because there is only one reference to any object; hence, the clean-up discipline is to reclaim resources when a variable goes out of scope. Additionally, exchanging assignment also simplifies specification and reasoning by unifying values and references in a fundamental way: there is no logical difference to the client between exchanging two object references and exchanging the values of the referenced objects. This means that introducing the exchanging assignment in place of preserving assignment — and in this case in preference to the other two possible data movement operators as well — facilitates a move toward a uniform value semantics. This is *precisely the opposite direction* taken by Java and .NET. Further details on the consequences of such a move are discussed in [Weide and Heym 2001].

8. Acknowledgments

This work was supported in part by the National Science Foundation under grants number CCR-0081596, and CCF-0811737. We also wish to thank other current and former members of the Resolve/Reusable Software Research Group, and especially Neil Coplin and Olga Volgin, for their useful comments and contributions.

References

Henry G. Baker. "Use-once" variables and linear objects: storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, 30(1):45–52, 1995. ISSN 0362-1340.

Anindya Banerjee and David A. Naumann. Representation independence, confinement and access control. In *Proceedings of the 29th POPL*, pages 166–177. ACM Press, 2002.

Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Proceedings of ECOOP 2002*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255. Springer-Verlag, 2002.

S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7:70–90, 1978.

James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Welsey, New York, NY, 1992.

George W. Ernst and William F. Ogden. Specification of Abstract Data Types in Modula. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 2(4):522–543, 1980.

George W. Ernst, Raymond J. Hookway, and William F. Ogden. Modular verification of data abstractions with shared realizations. *IEEE Trans. on Software Engineering*, 20(4):288–307, 1994.

Manuel Fähndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. In *Proceedings of PLDI 2002*, pages 13–24. ACM Press, 2002. ISBN 1-58113-463-0.

Douglas E. Harms and Bruce W. Weide. Copying and swapping: influences on the design of reusable software components. *IEEE Trans. on Software Engineering*, 17(5):424–435, May 1991.

Laurie J. Hendren and Guang R. Gao. Designing programming languages for the analyzability of pointer data structures. *Computer Languages*, 19(2):119–134, April 1993.

Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, Snowbird, UT, June 2001. URL `http://www.research.ibm.com/people/h/hind/paste01.ps`.

C.A.R. Hoare. Hints on programming-language design. In C.A.R. Hoare and C.B. Jones, editors, *Essays in Computing Science*. Prentice Hall, 1989.

C.A.R. Hoare. Null references: The billion dollar mistake. Presentation at QCon London, March 2009.

John Hogg. Islands: aliasing protection in object-oriented languages. In *Proceedings of OOPSLA '91*, pages 271–285. ACM Press, 1991.

John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva convention on the treatment of object aliasing. *ACM SIGPLAN OOPS Messenger*, 3(2):11–16, 1992.

Joseph E. Hollingsworth, Lori Blankenship, and Bruce W. Weide. Experience report: using RESOLVE/C++ for commercial software. In David S. Rosenblum, editor, *Proceedings of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE-00)*, volume 25 of *ACM Software Engineering Notes*, pages 11–19. ACM Press, 2000.

J. J. Horning. A case study in language design: Euclid. In F. L. Bauer and M. Broy, editors, *Program Construction*, volume 69 of *Lecture Notes in Computer Science*, pages 113–132. Springer Verlag, 1978.

R. B. Kieburtz. Programming without pointer variables. *ACM SIGPLAN Notices*, 11(3S):95–107, March 1976.

Jason Kirschenbaum, Heather Harton, and Murali Sitaraman. A Case Study in Automated, Modular, and Full Functional Verification. In *AFM '08: Third Workshop on Automated Formal Methods*, pages 53–58. ACM, July 2008.

Gregory Kulczycki and Jyotindra Vasudeo. Simplifying reasoning about objects with Tako. In *Fifth International Workshop on Specification and Verification of Component-Based Systems*, pages 57–64, New York, NY, USA, 2006. ACM.

Gary T. Leavens and William E. Weihl. Reasoning about object-oriented programs that use subtypes. In *Proceedings of the Joint ECOOP/OOPSLA Conferences*, pages 212–223. ACM Press, 1990.

K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 24(5):491–553, 2002.

Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

David C. Luckham and Norihisa Suzuki. Verification of Array, Record, and Pointer Operations in Pascal. *ACM Trans. on Prog. Languages and Systems (TOPLAS)*, 1(2):226–244, 1979.

Naftaly H. Minsky. Towards alias-free pointers. In *Proceedings of ECOOP '96*, volume 1098 of *Lecture Notes in Computer Science*, pages 189–209. Springer-Verlag, 1996.

D. Musser, G. Derge, and A. Saini. *STL Tutorial and Reference Guide, Second Edition*. Addison-Wesley, Reading, 2001.

James Noble, Jan Vitek, Doug Lea, and Paulo Sergio Almeida. Aliasing in object oriented systems. In *ECOOP '99 Workshops*, volume 1743 of *Lecture Notes in Computer Science*, pages 136–163. Springer-Verlag, 1999.

James Noble, David Holmes, and John Potter. Exclusion for composite objects. In *Proceedings of OOPSLA '00*, volume 35 of *ACM Sigplan Notices*, pages 13–28. ACM Press, 2000.

John C. Reynolds. Syntactic control of interference. In *Proceedings of the 5th POPL*, pages 39–46. ACM Press, 1978.

Murali Sitaraman. Building a Push-Button RESOLVE Verifier: Progress and Challenges. Technical Report RSRG-09-01, School of Computing, Clemson University, Clemson, SC, January 2009. URL `http://www.cs.clemson.edu/~resolve/reports/RSRG-09-01.pdf`.

Murali Sitaraman and Bruce W. Weide. Component-based software using RESOLVE. *ACM SIGSOFT Software Engineering Notes*, 19(4):21–67, 1994.

Sun. Java Platform, Standard Edition 6 API Specification. World Wide Web electronic publication, 2009. URL `http://java.sun.com/javase/6/docs/api/overview-summary.html`.

B. Weide, S. Edwards, D. Harms, and D. Lamb. Design and specification of iterators using the swapping paradigm. *IEEE Trans. on Software Engineering*, 20(8):631–643, August 1994.

Bruce W. Weide and Wayne D. Heym. Specification and verification with references. In *2001 OOPSLA Workshop on Specification and Verification of Component-Based Systems*. http://www.cs.iastate.edu/ leavens/SAVCBS/papers-2001, 2001.

B.W. Weide, W.D. Heym, and J.E. Hollingsworth. Reverse engineering of legacy code exposed. In *Proceedings of the 17th ICSE*, pages 327–331. ACM Press, 1995.

Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. *SIGPLAN Not.*, 43(6):349–361, 2008.