Clovers:
The Dynamic Behavior of Types and Instances

Lynn Andrea Stein
Stanley B. Zdonik

Brown University
Dept. of Computer Science
Providence, Rhode Island 02912

Technical Report No. CS-89-42
November 1, 1989

Clovers:
The Dynamic Behavior of Types and Instances

Lynn Andrea Stein and Stanley B. Zdonik*
Department of Computer Science
Brown University
Providence, RI 02912
401-863-7600
{las,sbz}%cs.brown.edu@relay.cs.net

Abstract

Clovers are a new mechanism for object-oriented languages that relax the constraints of the conventional type/instance distinction. Clovers provide a new definition of object-hood, in which a single object may consist of multiple overlapping representations, sharing aspects of both behavior and identity. We show how clovers can be used to implement multiple views, changes to the type of an object, and expanded type notions such as minimal template. We argue that clovers provide a useful unification of the type/instance relaxations that have been presented in the literature, such as versioning, prototypes, and boolean classes.

*The authors are supported by an IBM Graduate Fellowship, and grants from IBM, DEC, Apple, ONR, and US West, respectively.

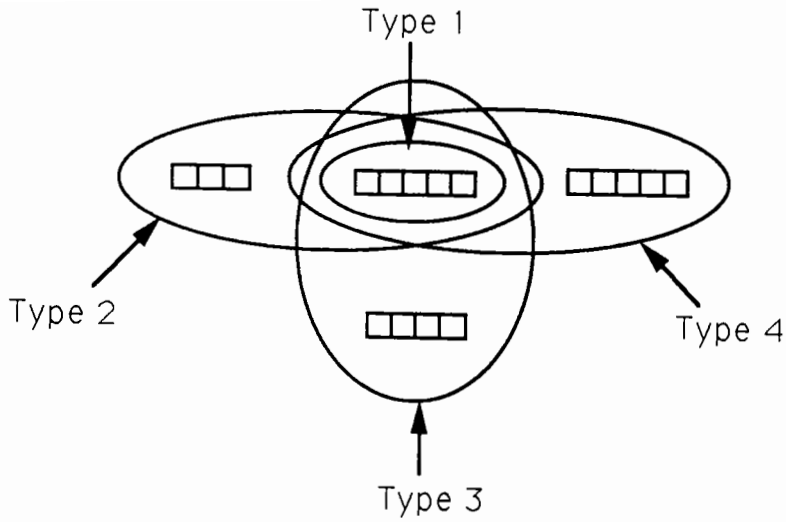

Figure 1: A clover contains a "chunk" for each component type.

1 Introduction

In most object-oriented languages, an object has a single defining "type," often corresponding to its class. This type determines the complete behavior of an object. It is typically not possible to modify this behavior after the object has been created. This view of types and objects simplifies the type systems of object-oriented languages, making type checking and type inferencing much simpler. However, it is often too narrow a view. It is frequently necessary to deviate from the strict behaviors that are provided by a type. It is also often important to be able to change the type of an object as that object changes, to limit our view of the object, or to treat an object as if it possessed the attributes of some other type. In this paper, we introduce a set of mechanisms that define *clovers*, objects that address these more flexible typing requirements.

Clovers contribute several new concepts to the type/instance assumptions of object-oriented languages. First, they allow many object representations to share the same identity. That is, it is possible to make sense of the statement that all of these instance representations are in fact the same object, although they may have different external appearances. Second, it is possible for all of these representations to share some portion of their state (i.e. internal variables) so that any modifications of that state will be reflected in all of the object representations. In this sense, clovers generalize existing sharing mechanisms such as prototypes. Third, they extend the traditional notion of coercion to include such concepts as versions, changing type, database views, and possible world semantics. Clovers thus provide a unifying framework for diverse ideas with widely different areas of application.

Intuitively, a clover is an object that contains several coexisting chunks of state for all of the "subtype pieces" of an object (see figure 1. For example, a toyota has a chunk containing the representation of its car variables and another chunk containing the additional representation for its toyota variables. We can represent two competing views of a car by a clover with a shared chunk for the car part, and two distinct chunks, each representing a competing toyota part. A reference to this clover would indicate which of the two toyota "views" it referred to.

Clovers can be used to simulate many useful extensions to typing that occur naturally in some applications. For example, we can use clovers to provide the behavior of versions, changing types of instances, views, and possible world semantics. We can also use the same mechanisms to introduce

such behaviors as minimal template, multiple class membership, and union type. In this paper, we will demonstrate the use of clovers to achieve some of these notions.

Clovers require the introduction of several operators for forming and referring to object variants. Some of these operators are familiar, resembling coercions in traditional languages. Others are new, and involve adding behavior to existing objects. At the heart of these coercions is the notion of *object reference*, which describes both a particular object and a particular "view" of that object. In the context of clovers, reference takes on a role similar to that of object identity. We discuss the ways in which traditional notions of equality must be expanded to account for these new notions of object identity.

The rest of this paper addresses these issues in more detail. In the next section, we describe some of the grounding of clovers in traditional programming language concepts. We then discuss the technical notion of clovers, and illustrate their use in some real examples. Finally, we compare this approach to other related work.

1.1 Some background

Several of the ideas described in this paper have roots in general and object-oriented programming language theory. In this section, we briefly discuss those elements of programming language theory that we use throughout the remainder of the paper: coercion, object reference, typechecking, and union type.

Many languages include the notions of automatic or explicit coercion of values. This is often seen in the operands to certain built-in operators like addition (i.e., $2.3 + 4 = 6.3$ with the 2 coerced to 2.0). In this case, the value of the coerced object is not changed. Instead, a new object is derived from the first. This differs from the "supertype coersion" of many polymorphic languages, in which a value of a more specific type can be assigned to a variable of a less specific type. Here, the reference (i.e., the description contained in the variable) is changed, while the value remains unchanged. As will be seen, our definition for coercion is very close to this view. Alternatively, coercion could actually mutate the original value. Our *become* operator has this effect.

Throughout this paper, we discuss the concept of an object *reference*. This is a combination of the object id, and the particular "view" of the object we're considering. For example, we may view a particular person object as a person, or as an employee, or as a student. Each of these "views" is a distinct reference to the object, although they all refer to the same underlying object. Assigning a value to a variable corresponds to giving that variable a reference: not just the object, but the particular view of the particular object. In traditional object-oriented languages, this reference can be the object id plus the type. In the new languages we describe, an object reference becomes more complicated.

We are interested in allowing as much static type-checking as possible in our type system. In order to perform static checking, it must be possible to reason about the types of all expressions, and to know precisely whether the type of an expression matches the predeclared type of a variable or a formal parameter to a function. We have designed our operators and our compound type expressions to ensure that they are compatible with static checking.

Union types are a well-known concept from many conventional programming languages. A union type is a declared type for a variable that states that the value that the variable is allowed

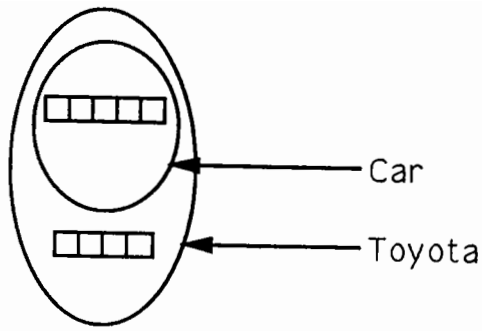

Figure 2: In most object-oriented languages, a toyota can be viewed as a car.

to assume is one of some finite collection of types. The variable can only refer to one of them at a time. It can be assigned other values with types from the list, but a given value does not change its type. That is, the use of union type is for variables not for values.

2 Technical discussion

2.1 Coercions up and down the hierarchy

In most object-oriented systems, we can treat an instance as an instance of a supertype—e.g. a toyota as a car—without requiring special mechanics. In this paper, we'll define the coersion "take a toyota, `t`, and return the car part" as

```
c := coerce-(Toyota,Car,t)

        where coerce-(FromType,ToType,obj)
                takes obj: FromType
                and returns : ToType
```

The function `coerce-` returns a new "view" of its input object. This toyota, for example, is still the very same toyota we started with, but the variable `c` now contains a view of the toyota *as a car*. The variable `t` still holds the whole "toyota view," as it did before. Figure 2 shows the toyota viewed as a car.

Although the car and the toyota in the last paragraph are both the same object, they correspond to different views of that object. In particular, while `t` and `c` refer to the same object, they contain different *object references*. An object reference (or just reference) combines both the object identity, and some notion of the particular "view" of the object we're considering. Assigning a value to a variable corresponds to giving that variable a reference: not just the object identity, but an indication of which view of that object we mean.

We've just described limiting the view of an object we have; we can also talk about going in the other direction. For example, if we have a "car view" of something that's really a toyota, we

may want to lift the "car-mask" and view it as a toyota again. This is just shifting references as we did before, but now we're enlarging our view:

```
        t := coerce+(Car,Toyota,c)

        where coerce+(FromType,ToType,obj) takes obj: FromType and
returns : ToType [ a subtype of FromType ]
```

Coerce+ can be a dangerous operation. If we try to restore the toyota behavior of a car that is *not* already a toyota, for example, we will cause (at the very least!) severe typing violations that cannot be determined at compile-time. One way around this is to use typechecking *guardians*, as in ALGOL-68 [ALG68], to ensure typing constraints: "if the object *has* a toyota part, coerce+ it." With guardians, and assuming that the type passed to coerce+ is compile-time determinable, the coerce+ operation and static typechecking are entirely compatible. If static typechecking is not required, guardians to coerce+ become unnecessary.

A more practical way to manage the power of the coerce+ operator is to require as one of its arguments a *key* to the part of the object to be restored. This key can be the object reference—the "name" of the view of the object. In this case, coerce+, like coerce-, becomes a convenient shorthand for manipulating these references. Of course, passing keys—or references—is not a practical solution if it requires that each object keep track of all of the possible ways it can be viewed. System-wide functions of the form "get me (all of) the toyota's this car is" may return a list of relevant keys; coerce- then uses such a key to locate the appropriate object reference. Such a system of keys is not in general statically typecheckable, but does add a certain amount of runtime information. By restricting access to such keys, the system can also control a user's view of a particular object.

2.2 Become: enlarging the object

In the previous section, we discussed shifting views, or references. Both coerce+ and coerce- create new references to existing objects. But what if the view we want is not available—the car in question has never been viewed as a toyota before? In this section, we describe the become function, which allows the user to specify the missing behavior, changing the underlying object as well as creating a new reference. Since the new information is outside all existing references, this change to the object is transparent: no other reference knows that this car is now a toyota.

Become takes an object and two types, as well as a specification of the missing behavior:

```
        t := become(Car,Toyota,c,(i))

        where become(FromType,ToType,obj,extras)
                takes   obj:  FromType
                        extras:  ToType - FromType        [ a set of objects ]
                and returns   :  ToType
```

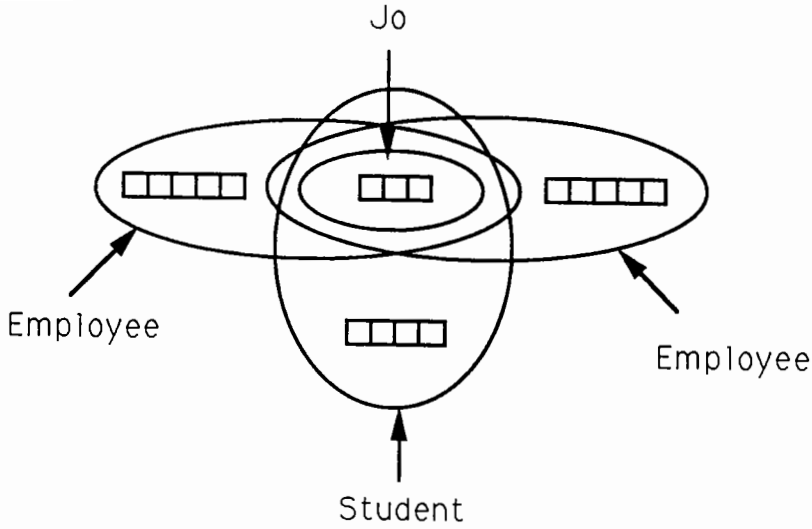

Figure 3: Jo is a person who is both a student and an employee.

The object returned by `become` is essentially a new view of `obj`, with a new type. In the same way that we can choose to regard a toyota as a car, this allows us to regard a car as a toyota, *provided that we specify the extra* (toyota) *behavior.*

The "type" `ToType - FromType` refers to those fields (e.g. methods and variables) of ToType objects not inherited from FromType. So, for example, `toyota - car` refers to fields, like importer, defined for toyotas but not for cars. However, no object can actually have a type like this: the type of an object must be "solid" (i.e. contain a complete path to the root of the hierarchy). The algebra of compound type expressions, described in appendix A, gives more details of this extended type expression language.

2.3 Clovers

The idea of allowing an object to "become" an instance of a subtype is not all that strange. However, it has some unusual ramifications. Consider, for example, an arbitrary person object, Jo. Suppose that we decide to extend this person to include information about her status as an employee. This can be done by providing information to fill all of the fields of employee that are not inherited from person.

But what if we now discover that Jo is actually a student. Now, we have three references to this object: Jo, the person; Jo, as an employee; and Jo, the student. At any given time—or for any single user—Jo may be regarded as an employee, or as a student, or neither. However, her employer—or anyone else who regards her as an employee—may know nothing about her grades or her course enrollment, while Jo's professors have no access to Jo's salary information. Figure 3 shows this complex object, which we call a *clover*.

This is analogous to the (more limited) situation in standard object-oriented languages, where we may choose to regard an object—say a toyota—as an instance of a subtype—e.g. car. Viewed as a car, this object lacks such information as its importer, even though this may be contained in a field of the toyota object that the car really is. However, in traditional object-oriented languages, all of these subclass views form a single total order of containments. These views represent larger or smaller pictures of the whole; and to see the whole, we need merely view the object as its most specific subclass.

In the clover model, there may not be a single most specific subclass. The branches (leaves) of the clover correspond to disjoint views, and examining Jo as a student may never provide insight into her status as an employee. To obtain access to another leaf of the clover requires the appropriate permissions or keys, and without them, the other leaves remain transparent: Jo the student is indistinguishable from a student without an employee leaf.

These keys to other branches of the clover are simply object references: they completely determine both the object, and the view of the object. They can, for example, be regarded as names for paths down the type hierarchy. Unfortunately, where in standard languages the object reference plus the class suffices to uniquely determine the scope of any given view, in clovers this is insufficient. For example, if Jo holds two jobs as well as going to school, we might extend her twice, in two become operations. In this case, "Jo's employee extension" will still be an ambiguous label. A complete identifier must contain both the unique object id and a leaf id which determines both the leaf, and which type along that leaf, is referenced.

2.4 Equality

In standard object-oriented programming languages, there are two concepts of equality. The first, which we will call *similarity,* obtains when two objects have the same "appearance." That is, they have the same fields, and the values in each field are similar; at this moment, the two objects are indistinguishable.[1] However, the two objects are not necessarily the same object, and so changes to one may not affect the other. Similarity is thus a property that holds at some time, but may vary over time. Our definition of similarity is slightly weaker than what [KC86] calls "deep-equal."

On the other hand, two objects may have the same id—i.e. they may be literally the same object. In this case, we say that the objects are *identical* ([KC86] also calls this equality "identity"). Identity and similarity are independent: two objects can be similar, but not identical, or they can be identical, but of different classes—e.g. a toyota may be viewed as a car, and the car and toyota views have the same identity, but are not similar. However, in conventional languages, identity implies similarity up to subclass: if two objects are identical, one is actually a subclass-view of the other.

When two objects are the same view of the same object—i.e., they have the same reference—we call them *equivalent.* In conventional languages, objects are equivalent precisely when they are similar views of identical objects. Thus, the car and toyota of the preceding paragraph are actually equivalent up to subclass—the car is actually a "subclass piece" of the toyota.

In the world of clovers, the definition of equality becomes more flexible. Similarity—"equality of appearance"—still applies. Similarly, each clover has only one object id, and only one identity. But it is no longer true that similar identical objects are equivalent: one car may be two toyotas—sharing object id, and therefore identical—each with the same values for fields—and therefore similar—but the two toyotas are not the "same view of the same object" equivalence requires. Figure ?? depicts such a situation. Equivalent objects are similar and identical, but the two separately are not sufficient to determine equivalence. Thus, in a clover language, all three equality operators are necessary.

[1] Here, indistinguishable really means "up to object ids"—if the objects have different ids, they can always be distinguished in that way.

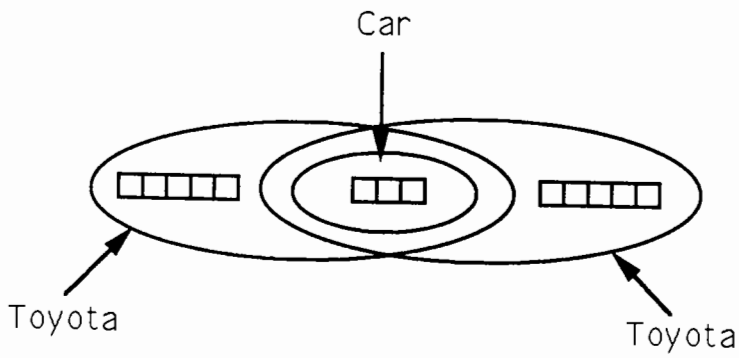

Figure 4: This car has two toyota extensions that are similar, and identical, but they're not equivalent.

3 Applications

3.1 Minimal template

The notion of a minimal template has been described in [Ste87] and [Ste89]. A minimal template is a type that defines common behavior of its instance, but allows its instances to add other operations to their behavior beyond those defined by the template. This is different from standard typing in the sense that normally a type defines all the operations that an instance can support.

Assume that a report is defined as follows:

```
Report = (super: Document;
          vars:
                    author:  String
                     title:  String
                  chapters:  Set [Chapter])
```

A report with a cover page would be created as follows:

```
r:        Report;
cover:    CoverPage;
myReport: Report + (CoverPage);

myReport := coerce+(Report, Report + (CoverPage), r, (cover));
```

The variable myReport will now contain an object of type `Report + (CoverPage)`. This type does not exist in the type lattice. The instance that is stored in myReport may well be the only instance of this type in existence. The type Report is in the type lattice and acts as the minimal template for the new instance. We add to this minimal template the additional operations and properties that are defined by CoverPage.

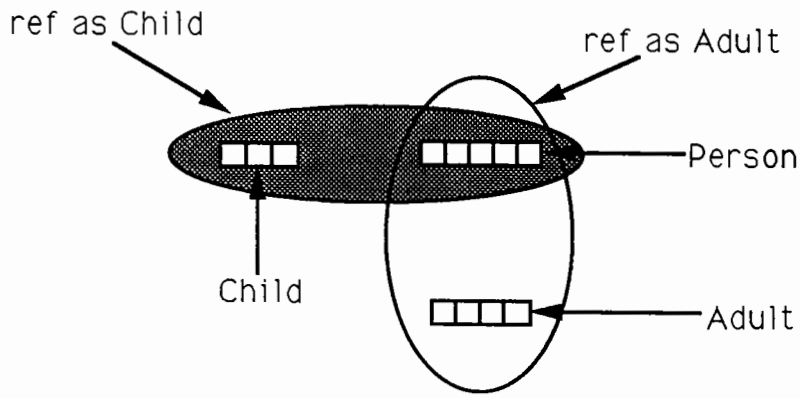

Figure 5: When a child becomes an adult, a new leaf is added to the clover.

Notice that it is still possible to have a reference to this object as just a Report. Moreover, it is possible to augment the Report part of the object by adding another clover leaf to the minimal template (i.e., Report) that adds an Appendix component. This would yield two different objects that share the same state for their Reportness.

3.2 Union types

Union types are types defined by disjunctions. They may be disjoint over time: a person grows from an infant to a child to an adult, and so on, but cannot be more than one of these types at the same time. The disjunction may also refer to views: a pocketknife, depending on the blade in use, may be a corkscrew, a screwdriver or a knife. Alternatively, the union may reflect the fact that the types of its elements are disjoint: furniture may be a table or a chair or a bookshelf, but an individual piece of furniture does not change its type. We will look at how each of these notions can be accommodated by clovers.

The first example, of a person growing older, is a good example of an object's changing type. A classic problem with this type of behavior [Zdo87] is that when an object changes its type, other objects or variables pointing to it may have assumed that their referents were of the original type. This "versioning problem" can lead to some very unexpected behavior.

The clover approach to this problem is that when we must change the type of an object, we simply grow a new path that includes the new type. Any previous references to this object will still point to the older path, which is still accessible. We would, therefore, have definitions like the following:

```
Person = (super: Animal;
          vars:
                  age:  Int
                  sex:  String
              parents:  Set [Person])

Child = (super: Person;
```

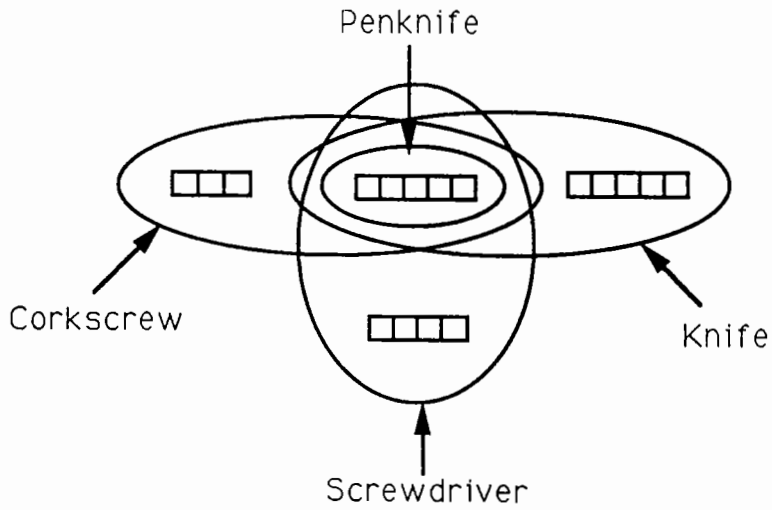

Figure 6: A pocketknife can be a screwdriver or a corkscrew or a knife.

```
                vars:
                      school:  String
                     teacher:  Person)

Adult = (super: Person;
                vars:
                    employer:  String
                      salary:  Int)
```

If a Child object changes its type to an Adult, we would grow another leaf on the clover with an extra piece of state for the Adult type: grow-up is defined by

```
        Me-as-adult := become(Adult,Person,Me,(MyEmployer,MySalary))
where
        Me := coerce-(Person,Child,Me-as-child)
```

The leaf containing the state for the Child type is still available through the references that continue to point to it. Figure 5 depicts this clover.

The pocketknife example, which is illustrated in figure 6, can be handled by having a clover with three branches, one for the Corkscrew type, one for the Screwdriver type, and one for the Knife type. This kind of union type is actually a "clover type": every instance of type Pocketknife will be *created* as a clover, with corkscrew, screwdriver, and knife blades (or branches of the clover) in place. The "body" of the pocketknife will contain, among other things, the appropriate references to these branches, and a special `switch-blade` method. Thus, a corkscrew blade can turn into a screwdriver at runtime, by switching references to another leaf of the clover. Methods that affect the whole pocketknife—such as `move`—will affect each blade/leaf, but while those particular to a single blade—such as uncork—require that that blade be in place:

```
a-pocketknife: PocketKnife

a-corkscrew := coerce+(Corkscrew,PocketKnife,a-pocketknife)
```

```
a-screwdriver := coerce+(Screwdriver,PocketKnife,a-pocketknife)

move (a-corkscrew, tabletop)
where-is (a-screwdriver)   ->   tabletop
remove-cork (bottle, a-corkscrew)
```

In this example, the invocation of the **move** operation on the corkscrew blade changes the position of the screwdriver blade as well, but **remove-cork** requires that its second argument be of type Corkscrew.

Disjoint types, such as the furniture described above, are also "clover types," but each instance of the type only has a single branch. For example, a piece of furniture can be a table or a chair, but instantiating **furniture** doesn't mean creating a table-chair clover. These types are similar to the "or-classes" described in [MZ86], but have exclusive-or properties. The relation between such exclusive-or types and existing notions of types bears further investigation.

4 Comparisons with other work

4.1 Multiple inheritance

Both multiple inheritance and clovers provide useful behavior to an object-oriented type system, and in general, both are necessary mechanisms for an object-oriented language. While both allow objects to have more than one type, they accomplish this in quite different ways.

Multiple inheritance is a static, atemporal mechanism: an object is, at its creation—and ever after—an instance of all of its superclasses. In contrast, while the "whole" clover belongs to several different classes, each branch, or view, of the clover is (in general) a member of a single most-specific class. A particular reference to a clover has only one type at any given time. In this sense, clovers are more limited than multiple inheritance. But a variable containing a reference to a clover may change its view over time, without changing the identity of the object it points to. In this sense, the type of a clover may change over time, while that of a multiple-inheritance instance may not.

Consider the PocketKnife class of section 3.2. A pocketknife could be defined as a multiple-inheritance subclass of Corkscrew, Screwdriver, and Knife. In this case, an instance of PocketKnife would be all three of these things at once. The only way to select a particular blade would be to `.coerce-` the pocketknife to mask out the other blades. To switch to another blade, we'd need to `coerce+` back to the original multiple-inheritance instance, and then to `coerce-` to the new blade. And multiple inheritance cannot rule out a variable that contains references to two or more blades of the pocketknife at the same time. In contrast, the clover representation of a pocketknife captures exactly those properties we'd expect such a "union type" to have.

4.2 Versions

Many proposals have been produced recently to include version mechanisms into object-oriented databases. An object is viewed as a partial ordering of versions that each record a snapshot of the

object in time. It is a partial order since branching of alternatives is often allowed. This type of behavior is possible to simulate with clovers, but clovers are much more flexible. For example, it is not possible with standard version schemes to allow two different versions to share part of their state such that a change to one version object is reflected in another. Also, all versions of an object are typically of the same type. Clovers allow both shared object state and multiple paths with different types.

There has also been some interest in allowing objects to change their types over time. Some of the problems that this raises have been described in [Zdo87]. Clovers treat this problem by allowing an object to extend itself to be of another type, but older references to the object can still see the object as it was when it was of the previous type. This avoids many of the difficult typing problems, but allows the same object to be treated as several types at once. This suggests that there might need to be a mechanism to allow a program to inquire about the other branches of a clover.

4.3 Actors and prototypes

Our use of the "become" operator may invite comparisons with the actors languages [Agh87]. Such comparisons are apt, and there are similarities in the ability of an object to transform itself, particularly in the example of "clover types" such as the pocketknife. However, when an actor "becomes" a new object, the old object is replaced by the new. In contrast, the "become" operator of clovers is neither a replacement, nor the creation of an independent object. The new actor functions independently of the old, now defunct, actor. But the new clover leaf is only a leaf on an existing object: it shares identity and a certain amount of behavior.

On the other hand, the actors languages have also generated the idea of creating new objects from other objects, rather than from types. This idea has been further expanded by [Lie86] and others and now has life outside the actors model. Clovers share some of this flavor. However, when a prototype is extended, an entirely new object is created, This object has its own separate identity and existence, although it may share some of the behavior of its prototype. In contrast, when a clover is extended—a new leaf is added—the object identity remains the same. This is stronger than the sharing of methods and variables: some part of the new object actually *is* the original object.

5 Conclusion

Clovers identify a new view of the composition and dynamic behavior of objects. We believe that this view solves solves some of the previous problems with the conventional type/instance relation put forth by most object-oriented languages. Clovers make it easy to describe objects that change their types, multiple views of an object, and hybrids of the standard type/instance mechanism and prototypes such as minimal template.

A clover has the following important property: it allows many logical object to share a piece of their state as well as some aspects of their identity. It is possible to construct multiple references to a clover object, each viewing a different part of that object. These views can be of different types;

they may share some behavior, and have other behavior local to the view. We have discussed the effects of clovers on object identity and equality, showing that traditional notions can be extended in a straightforward manner to allow for clover mechanisms, and we have described several realistic situations in which the flexibility provided by clovers allows natural models of the task at hand.

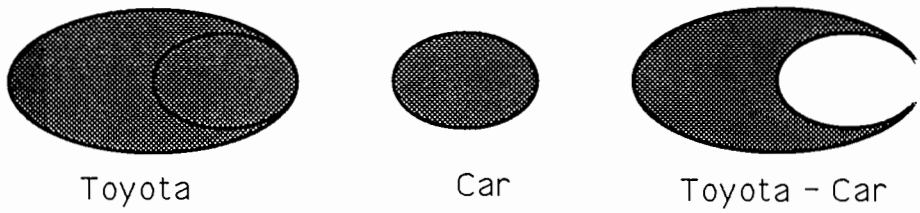

Figure 7: `Toyota - Car` refers to the extra piece of toyota behavior that cars don't have.

A Compound type expressions

A compound type expression (CTE) is an expression of the form:

- $\mathcal{T}$, where $\mathcal{T}$ is a preexisting type
- $\mathcal{T}_1 + \mathcal{T}_2$, where $\mathcal{T}_1$, $\mathcal{T}_2$ are CTEs
- $\mathcal{T} + \{\text{op}_1, \ldots, \text{op}_n, \text{var}_1, \ldots, \text{var}_m\}$, where $\mathcal{T}$ is a CTE and $\text{op}_i, \text{var}_i$ are methods and variables defined according to the method/variable definition rules of the language
- $\mathcal{T}_1 - \mathcal{T}_2$, where $\mathcal{T}_1 \mathcal{T}_2$ are CTEs

Equivalence of CTEs follows the usual rules of associativity, commutativity, and scoping for $+$ and $-$:

$$
\begin{aligned}
(\mathcal{T}_1 + \mathcal{T}_2) + \mathcal{T}_3 &= \mathcal{T}_1 + (\mathcal{T}_2 + \mathcal{T}_3) \\
\mathcal{T}_1 + \mathcal{T}_2 &= \mathcal{T}_2 + \mathcal{T}_1 \\
\mathcal{T}_1 + (\mathcal{T}_2 - \mathcal{T}_3) &= (\mathcal{T}_1 + \mathcal{T}_2) - \mathcal{T}_3 \\
\mathcal{T}_1 - (\mathcal{T}_2 + \mathcal{T}_3) &= (\mathcal{T}_1 - \mathcal{T}_2) - \mathcal{T}_3
\end{aligned}
$$

$\mathcal{T}_1 + \mathcal{T}_2$ is intuitively the "multiple inheritance subclass" of $\mathcal{T}_1$ and $\mathcal{T}_2$. $\mathcal{T}_1 - \mathcal{T}_2$ refers to the "extra" behavior $\mathcal{T}_1$s have but $\mathcal{T}_2$s don't. This makes sense only when $\mathcal{T}_1$ is a subclass of $\mathcal{T}_2$ (or the expression can be rewritten to make this property true). This requirement is captured by "solidity": A *solid* CTE is one which has an equivalent CTE without $-$. A *hollow* CTE is one which is not solid; i.e. cannot be rewritten without $-$.

Compound type expressions may be used to construct either new types or novel instances of preexisting types. However, to preserve type consistency, no object may have a hollow compound type. Negative CTEs, therefore, are useful only in the construction of a new object out of pieces of preexisting objects. For example, `toyota - car`, as in figure 7, refers to the extra piece of "toyotaness" that cars don't have, and is useful in adding toyota behavior to cars.

Compound type expressions can be used in a statically type-checked language, provided that they meet certain requirements. No single object can have a hollow type, although sets of objects, such as the argument **extras** to **become**, can have this type. In their strictest form, CTEs may be used to construct objects, but all actual objects constructed must have predefined types (i.e. the "pieces" must sum to a predefined whole, as the toyota in the example).[2]

[2] Allowing objects of non-predefined types permits *anonymous types*, which necessarily cannot be integrated into a typing system without classification mechanisms.

References

[Agh87] Gul Agha. *Actors.* MIT Press, Cambridge, Massachusetts, 1987.

[ALG68] Algol-68, 1968.

[KC86] Setrag N. Khoshafian and George P. Copeland. Object identity. In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 406–416, Portland, Oregon, September 1986.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 214–223, Portland, Oregon, September 1986.

[MZ86] David McAllester and Ramin Zabih. Boolean classes. In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 417–423, Portland, Oregon, September 1986.

[Ste87] Lynn Andrea Stein. Delegation is inheritance. In *Proceedings of the Second ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 138–146, Orlando, Florida, October 1987.

[Ste89] Lynn Andrea Stein. Towards a unified method of sharing in object-oriented programming. In *Workshop on Inheritance Hierarchies in Knowledge Representation and Programming Languages*, Viareggio, Italy, February 6–8 1989.

[Zdo87] Stanley B. Zdonik. Can objects change type? can type objects change? In *Proceedings of the Workshop on Database Programming Languages*, Roscoff, France, August 1987.