A RIGOROUS MODEL OF OBJECT REFERENCE, IDENTITY, AND EXISTENCE

William Kent
Database Technology Department
Hewlett-Packard Laboratories
Palo Alto, California
kent@hpl.hp.com

June 1991

CONTENTS:

ABSTRACT

The essence of identity is the determination whether two references are to the same thing. A formal model of reference and identity can be based on the denotations of occurrences of symbols in a computational system. The model applies to explicitly created objects as well as mathematical abstractions.

1 INTRODUCTION

Object identity is a pillar of object orientation. It is often described in terms of data structures [Khosh86,Atkin89], which then depends on implicit assumptions about the existence and distinctness of such structures. A more explicit behavioral treatment of identity, which does not depend on characterizing objects as data structures, can be based on the denotations of occurrences of symbols in a computational system.

Focus of attention is shifted from the objects themselves to the references to the objects. The model of identity then applies equally to things which are considered to be inside or outside the computational system. It also applies equally to explicitly created objects and to eternally existing abstractions like numbers and extensional sets.

Section 2 and Section 3 explore the problem and the approach, leading up to the formal treatment in Section 4 through Section 6. Section 7 discusses some implementation considerations. It's not until Section 8 that we confess to avoiding the question of what an object is in the first place.

2 WHAT'S THE QUESTION?

The central question of identity shouldn't be posed in terms of whether two objects are the same. In the first place, sameness is ambiguous (doesn't always mean the same thing); it sometimes signifies similarity, as in identical twins. Beyond that, if we know they are two objects, they aren't the same object. A predicate of the form *Identical*(O_1,O_2) is hard to explain if O_1 and O_2 are meant to be the objects themselves. Can the same object actually be present in the first operand and also the second? This question is best recast into determining whether two references are to the same object. O_1 and O_2 are references to objects, and we want to know whether they refer to the same object. (We'll state that more precisely below.)

Identity might be tested in terms of propagated effects: if changing a property of O_1 automatically changes the same property of O_2, then O_1 and O_2 refer to the same object. Such a test can be unreliable, in the presence of inheritance or derivation rules which automatically synchronize certain properties of distinct objects. If my manager is always my department's manager by definition, that doesn't make me and my department the same.

If all properties of the referenced objects seem to remain the same under all tests, it might be inferred that O_1 and O_2 refer to the same object. However, if {} is a constructor of extensional sets of objects, then it constitutes the ultimate test: the cardinality of the set of objects constructed by $\{O_1,O_2\}$ must be one.

Object identity might be predicated on creation, with distinct objects arising from distinct creation events. But some things are not subject to creation (it's certainly arguable whether such things are objects). And preexisting objects might have to be attributed to hypothetical creation events in the past, perhaps difficult to distinguish. In other cases, several creation events might be associated with the same object.

Durability and distinguishability are two important aspects of identity. Durability is relevant in the sense that a thing is still the same thing in spite of changes. There's some essence of it that endures through change. Which means there is an "it."

Distinguishability means we know this thing is not that thing, even if we don't know any distinguishing properties. Physical things might be differentiated by their location in space, but that doesn't help with abstractions. Distinguishability comes down to counting: knowing whether there is one thing or two. If we're asking whether there's one or two, we can't be dealing with the objects themselves, or we would know. We must be dealing with references to objects — "this" and "that" — wondering if they refer to the same thing.

3 SURROGATES AND REFERENCES

Information is full of references to things. References need to be reliable, not affected by changes in the properties of things. It should be possible to determine whether or not two references are to the same object.

The essential principle is than an object retains its identity despite changes to any of its properties. What does that mean in a computational system? What is the “it”? The direct solution is to provide something in the system which serves as a surrogate for the object.

An object is described in many object models as a chunk of storage containing data about the object. The chunk of storage can serve as a surrogate for the object (an electronic circuit or a person); it might even be the object itself (a stack, file, table, or tuple). Whether it is a surrogate or the object itself, references to the object serve directly or indirectly as pointers to that chunk of storage.

This approach isn’t always adequate [Khosh86]. The object might not be implemented in a contiguous chunk of storage, being fragmented on a small scale among files or tables or on a large scale distributed around a network. Its “address” is more of an abstract label than a memory pointer. If there are copies of the object, it may have several candidate addresses; references that appear different might arguably refer to the same object. Objects having no state might not have a chunk of storage to serve as surrogate. Numbers, strings, and even mathematical sets might be so treated; sometimes there is confusion between the object and the reference to the object.

What remains relatively stable is the notion of reference. Whether or not it can be mapped to a storage address, an object can have a data value which reliably refers to it. This is the essential notion of an “object identifier” (oid), although that term has come to mean a somewhat implementation-dependent construct of fixed format and length.

Oid’s themselves serve as surrogates for objects [Hall76,Khosh86]. Wherever one occurs, it stands for the object it identifies. The “oneness” of an object is implicit in an assumed one-to-one correspondence between oid’s and objects. Wherever that oid occurs, it stands for that object; if it’s a different oid, it’s a different object. Other, possibly mutable, properties are associated with the oid. The oid is the “it” which remains invariant.

Representation of an object’s properties involves some sort of associations between the object’s oid and its property values, which might be implemented in a variety of configurations. If properties of an object are implemented in a contiguous chunk of storage with the oid factored out, that coincides with the first view.

Some difficult questions still arise. One, as mentioned, is whether the referenced object (referent) is itself in the system. Closely related is the question of whether the existence of an object is connected to the allocation and deallocation of storage space for its properties. Sometimes we can’t tell the difference between the reference and the referent, as with numbers and extensional sets. We get confused between various notions of sameness, equality, and identity; identical twins are not the same person. We lose track of the difference between making copies and creating objects; between creating an object and referring to one. Does copying a memo make a new object? What about copying a string or a tuple? Does set construction create a new object? Other questions arise in coordinating multiple naming scopes, and with synonymous oid’s.

Such questions can be addressed in a rigorous model of object identity in a computational system. Such a model also clearly delegates responsibility for certain questions elsewhere. For example, it is not the model’s responsibility to determine whether two ideas are to be treated as the same object. That decision must be made externally, and object identifiers assigned accordingly. The computational model will thereafter treat the objects correctly.

Certain distinctions are carefully made in the computational model to avoid ambiguities underlying some common difficulties.

4 THE COMPUTATIONAL SYSTEM

A computational *system* is embedded in a *universe* of things both concrete and abstract (Fig. 1).

4.1 Symbols and Tokens

A computational system contains *tokens*, which are occurrences of *symbols*. The emphasis is on occurrences; symbols themselves are abstractions outside the system. In Figure 1, the two occurrences **@SAM999** are tokens in the system. They are occurrences of the symbol *@sam999* outside the system.

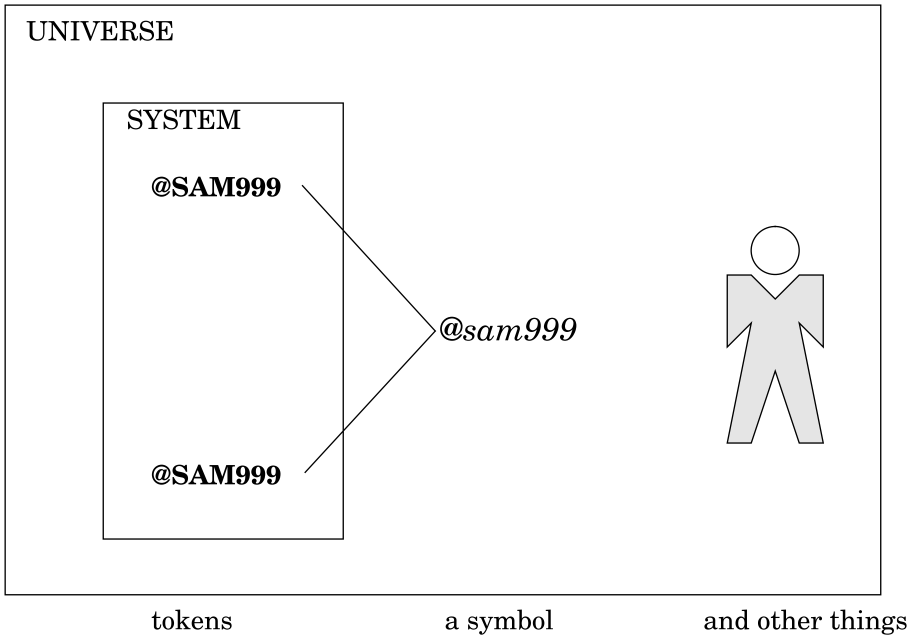

Figure 1.

The mapping

Sym: Token $\rightarrow$ *Symbol*

is a total (and single-valued) function from tokens to symbols outside the system. Not every symbol has a corresponding token in the system.

The intent here is that “object” and “object” are two tokens, each being an occurrence of the same symbol, which is an abstraction different from either of those occurrences. This is much like the copies of a book in a library. Each copy has a definite location, and each is a distinct entity from the abstract book of which it is a copy. The abstract book has no identifiable location in the library, and it exists whether or not there are any copies in the library.

The important thing about symbols is that they constitute a fixed though infinite set. They are the things which can possibly have occurrences in the system. They always exist; the population is invariant. They could be the set of strings over some alphabet, though they need not be considered readable signs in any sense. They exist in some abstract portion of the universe where *the* English alphabet, *the* letter x, *the* number 1, *the* empty set, *the* American flag, and Christmas each exist as a single concept.

In contrast, the set of tokens in the system varies. They are the things actually occurring in the system, and each occurrence is distinct. Change is inherent in the nature of a computational system, and the essence of the change is the appearance and disappearance of tokens, i.e., occurrences of symbols. Every token is distinct from every other token. Every token is an occurrence of exactly one symbol, though many tokens might be occurrences of the same symbol. In fact, the equality predicate over tokens $t_1=t_2$ is defined to mean that the tokens are occurrences of the same symbol.

The appearance of a new token in the system could be considered an act of creation, but it does not create a new symbol. Whether there are or have been other occurrences of the symbol in the system is immaterial.

Though not essential, it might be useful to think of tokens being differentiated by a location property, with no two tokens having the same location. (To be precise, tokens don't overlap. If strings can overlap in an implementation, then their location might best be modeled as a combination of address and length.) Token replacement means that a token having a given location is removed from the system, and a different token having that location is added.

Tokens exist outside the system as well. A token might be simultaneously inside and outside, i.e., at an interface to the system. In addition to strings, symbols and their token occurrences might also be icons or other images, possibly inside or outside the system, or at the interface.

Symbols have identity. They are distinguishable from one another, and they don't change. Occurrences of symbols in the system might be replaced by occurrences of other symbols, but the symbols themselves don't change. Symbols can therefore model the identity of other things. A symbol can correspond to the identity of one thing, and serve as a surrogate for that thing.

We are recapitulating the fundamental notions of language. To be very precise, we distinguish between tokens, symbols, and meaning. The tokens 10 and 10 (I omit quotation marks for now) are occurrences of a certain symbol which I can't write down, since the only things I can write are tokens. Furthermore, both the symbol and the token are different from any possible meaning they might have, such as the number of toes I have. Let me re-emphasize: for now, when I write 10 I am writing an occurrence of a symbol (it should really be quoted). When I want to mention the quantity of my toes, I will use words: I have ten toes.

The essence of language, and symbolism, is meaning. Symbols mean things; each occurrence of a symbol means what the symbol means. Denotation is a mapping from symbols to things (possibly including symbols and tokens: they are all things). In general, denotations are complex. Symbols have multiple meanings, often varying from one context to another.

We simplify things for the computational system. We start with symbols having a fixed, single denotation. Later we will introduce a rudimentary notion of scope, and some rudimentary synonyms.

A *handle* is a symbol having a single meaning, with no two handles having the same meaning. The mapping

Referent: Handle → Thing

is a single-valued and singular (1:1) function from symbols to things in the universe. It is a partial function on symbols, being defined only on handles. We can imagine a Valid predicate which differentiates handles from other symbols.

A *reference* is an occurrence of a handle in the system; it is a token. A handle refers to one thing; each reference which is an occurrence of that handle refers to the same thing, which defines the mapping

Referent: Reference → Thing

The Referent mapping is thus overloaded, being defined both on symbols and tokens:

Referent(token) ::= Referent(Sym(token)).

The Valid predicate is similarly overloaded:

Valid(token) ::= Valid(Sym(token)).

In Figure 2, the two occurrences **@SAM999** are references in the system. They are occurrences of the handle @*sam999* outside the system. All three of these have the human being as their referent, in some given scope.

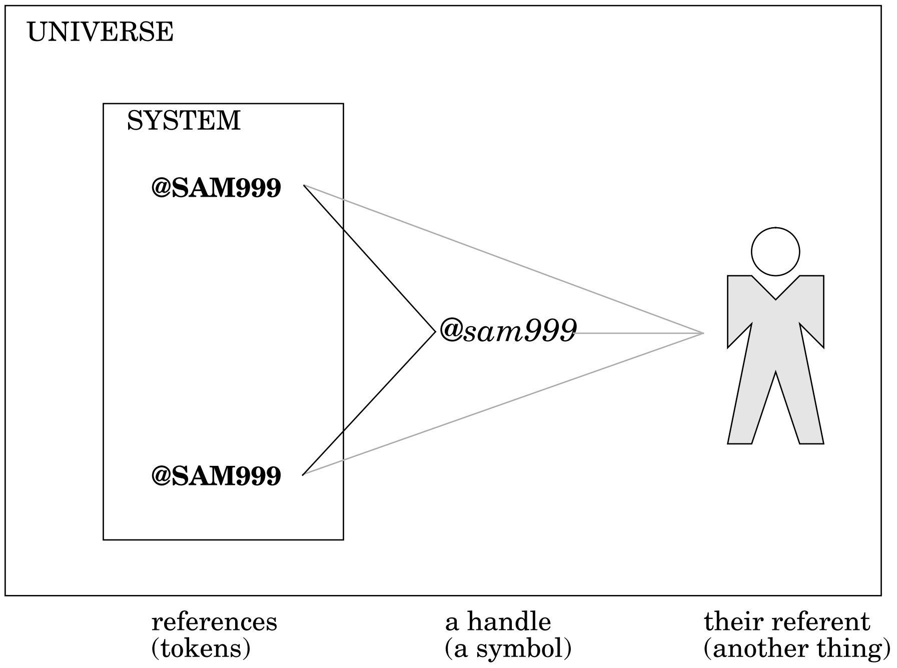

Figure 2.

The things denoted by a handle may be concrete or abstract. They might be things considered to be outside the computational system or inside, such as files, tuples, and stacks. The referents could even be symbols themselves. The token @billkent might be a reference to me. The token 10 might refer to the quantity of my toes (in a scope where we assume decimal notation). The token “10” (a four-character token) might be a reference to a certain two-character symbol.

Deciding which things are the same is very carefully excluded from the model. The model will not determine whether the morning star and the evening star are the same thing. The model will not determine whether the integer one and the real number one are the same thing. The model will not determine whether matching strings at different disk or memory locations are the same thing. Such questions are decided by system implementers or data administrators, expressed in the manner in which handles are defined.

4.2 Operations

The computational system is capable of executing operations whose operands and results are tokens. Computational events create new tokens, i.e., occurrences of symbols. They do not create new symbols, nor do they create things denoted by such symbols.

Information in the system is manifested in operations. The containment of a diagram in a document might be known through a *Content* operation. When applied to a token referring to the document, it returns a token referring to the diagram. Operations can also alter the behavior of other operations. An operation that removes a diagram from a document alters the results returned by the *Content* operation.

The *Sym* mapping is not an executable operation in the system, since its result is a symbol outside the system. Token equality is executable, implemented as an undefined primitive; we can only assume that equal tokens are occurrences of the same symbol.

The *Denotation* and *Referent* mappings are not executable in the system, since their results are often outside the system. The *Valid* predicate for tokens, on the other hand, could be implemented as an executable operation (assuming Boolean values are representable by tokens).

It will also be significant later that an operation is executed within one scope.

5 IDENTITY

At this elementary level, the model takes a very simple approach: two references refer to the same thing if and only if they are occurrences of the same handle, i.e., if and only if the tokens are valid and equal. If storage addresses (pointers) are used to implement references, then things at different addresses are different objects.

Sameness is very carefully distinguished from similarity and similar predicates. Our notion of equality has only been applied to tokens which are occurrences of the same symbol. Whether two tables or files or tuples match each other as character strings or by some other criterion is immaterial to our treatment of them as the same object. We even exclude the aspect of identity connoted by the phrase “identical twins”. So long as each person is denoted by a distinct handle, they are not the same object.

Beyond this elementary level, the model can be enriched by considering multiple scopes and synonymous handles.

5.1 Scope

The interaction of real systems involves independent oid generators and formats. The status of a symbol as a handle, and what it denotes, may vary from one scope to another.

Every operation is executed within a *scope*, which serves as an implicit parameter. Whether a symbol is a handle, and what it denotes, depends on the scope. If we were to make the implicit scope parameter explicit, the *Valid* predicate and the *Referent* mapping would be described as

$Valid: Symbol \times Scope \rightarrow Boolean,$

$Referent: Symbol \times Scope \rightarrow Thing.$

A given symbol might not be a valid handle in all scopes, or it might denote different objects. The same object might be referenced by different handles in different scopes.

A token belongs to a scope, which determines its status and meaning as a reference, based on the status and meaning of its corresponding symbol in that scope. The *Valid* predicate is defined on tokens, being true if the token is an occurrence of a symbol which is valid in the token’s scope. The token is then a reference, and its referent is the referent of the symbol in that scope.

In summary:

$Sc: Token \rightarrow Scope,$

$Valid(token) ::= Valid(Sym(token),Sc(token)),$

$$Referent(token) ::= Referent(Sym(token),Sc(token)).$$

Each scope might have its own mechanism for establishing the validity of symbols as handles, corresponding to the notion of independent oid generators. A database is likely to correspond to a scope. Scopes could be nested, but that's beyond the concern of this paper.

5.2 Synonymous Handles

Handles in different scopes might be synonymous, denoting the same object even if they don't match as symbols. To meet the needs of real systems, the model can also be extended to accommodate synonymous handles within one scope. If basic data types are considered to be handles, then the same abstract number or string might be denoted by different handles. The same number might be represented differently in different data types; there are various conventions for establishing the length or extent of a string in its representation. Synonyms also arise if it is possible to assert that several creation events actually introduced the same thing into a scope, and the several handles so validated denote the same thing.

With synonymous handles, the *Referent* function is no longer singular; different symbols might map to the same thing. Tokens t_1 and t_2 might have the same referent even if $t_1 \neq t_2$. Determination of object identity then becomes more complicated, relying on mechanisms generally outside the concern of this formal model. Object identity can be described in terms of a generalized Identity predicate

$$t_1 \equiv t_2$$

intended to be true if and only if t_1 and t_2 are valid references to the same object within the scope in which the identity is being tested. In the absence of synonymous handles, $t_1 \equiv t_2$ if and only if $t_1 = t_2$ (and they are valid).

While the formal model is not concerned with the algorithm underlying the *Identity* predicate, it cannot be entirely capricious. There are certain semantics of identity which should be observed. If $t_1 \equiv t_2$ is true in a scope:

- t_1 and t_2 must both be valid handles in the scope.
- t_1 and t_2 should not have been rendered valid by distinct creation events, unless they have been explicitly coerced to denote the same object.
- If there is an extensional object set constructor {}, then the cardinality of $\{t_1,t_2\}$ must be 1.

If t_1 and t_2 denote the same number, extensional set, or other such abstraction, then $t_1 \equiv t_2$.

The *Identity* predicate provides a basis for distinguishing between object-based and value-based operators. An operator f is *object-based* if replacing t_1 with t_2 in $f(t_1)$ would not change the result whenever $t_1 \equiv t_2$. Otherwise f is *value-based*. Value-based operators are concerned with and sensitive to the representations of object references.

It is possible to refer in one scope to the meaning of a symbol in another scope. Scopes can themselves be treated as denotable objects. In a certain initial scope, p_1 and p_2 might refer to two other scopes. An operator of the form

$$RemoteIdentity(t_1,p_1,t_2,p_2),$$

when executed in that initial scope, might be true if the meaning (referent) of t_1 in scope p_1 is the same as the meaning of t_2 in p_2. There might also be a corresponding operation to coerce two handles into denoting the same object:

$MakeSameObject(t_1,p_1,t_2,p_2)$.

This could be a basis for dealing with external or remote or local/global identifiers.

Whether or not these various identity operators are implemented, they serve to characterize the intended semantics of identity.

6 OBJECT EXISTENCE

How do we know that a thing exists?

In our terms, that's a metaphysical question, asking about something which may be outside the system. We're not even sure exactly what existence means. Within the computational system, the closest we can come to an operational description is in terms of handles.

All operations take tokens as operands. A few operations are really intended to operate directly on the symbols of which the tokens are occurrences. Such value-based operations might return the length of a symbol, or construct a certain token at a certain location, or indicate whether a symbol is a valid handle in a given scope.

Object-based operations, however, are intended to model information about the things denoted by the symbols, and cannot be meaningfully executed if the operand tokens do not refer to anything, i.e., are not valid handles.

The validity of a symbol as a handle becomes the crucial point. If a symbol is a valid handle, we have to assume that the thing it denotes exists. If we can even imagine the thing, if we can even have it in mind to ask whether it exists, then it exists. To be very precise, the computational system expresses an awareness of things, rather than the existence of things.

6.1 Creation

How do symbols become valid handles?

Certain symbols are always valid handles in some or all scopes, and are typically recognized by syntactic means. Examples include representations of mathematical abstractions such as numbers. Symbols whose intended denotations are other symbols are typically recognized by syntactic conventions such as quotes: the six-character token "%a4Z" is intended to denote a certain four-character symbol. In most implementations, recognizable representations of numbers and character strings are embedded in their handles.

Mathematical (extensional) sets, whose identity is determined by their membership, could similarly be referenced by syntactically recognizable handles, so long as their member handles were recognizable. The handle of an extensional set would logically behave as though it contained the handles of its members in some canonical ordering.

Other things which don't have natural, syntactically recognizable handles need to have arbitrary handles assigned to them. A pool of "invalid" (i.e., non-handle) symbols has to be reserved for this purpose, relative to a given scope.

Object creation renders valid a previously invalid symbol in a scope. It alters the behavior of the *Valid* predicate. Secondary activities in many implementations include registering new objects as instances of types or classes, allocating storage space for maintaining information about the objects, and initializing some of that information. Those are immaterial to the essential concepts of existence and identity.

Object destruction, if supported, renders invalid a previously valid handle. If handle re-use is not supported, the handle must further be rendered non-reusable. References involving the handle are no longer valid. Recovering storage space occupied by such references is a secondary and implementation-dependent activity, which might be done when the object is destroyed or when the reference is used.

Except for the possibility of reusing handles of destroyed objects, distinct creation events in a given scope validate distinct handles.

Intensional sets, whose identity cannot be determined from their variable membership, must explicitly be created and given arbitrary handles. This illustrates the more general concept of *carrier* objects, which are created to give persistent identity to something having mutable content. The carrier is immutable, while its *cargo* can be a different token at different times. We could imagine a *Cargo* operator which takes a carrier as operand and returns its cargo as result. Thus the cargo of an intensional set refers to an extensional set. A file or a document might be a carrier object whose cargo refers to a text string. As with intensional sets, distinct carrier objects might have cargos referring to the same thing. Thus “shallow equality” refers to the identity of the carriers, while “deep equality” refers to the identity of their cargos.

Strictly speaking, object creation doesn’t necessarily create anything. Creation, like any other operation, occurs within a scope. Its effect is to introduce an awareness of something into that scope. A previously meaningless symbol in that scope now has some meaning.

In terms of denotation, as seen from outside the system, it may be intended that things introduced into different scopes by distinct creation events are really the same object. Even worse, it may be discovered that the same object has been introduced several times into the same scope. There may thus be an operation intended to assert that the corresponding handles should be recognized as having the same denotation.

6.2 Construction

Operations in the system construct tokens, i.e., occurrences of symbols at locations. They do not create symbols, nor do they create the things the symbols might denote.

Executing 2+2 does not invent the number four, nor does it even invent the symbol “4”. All it “creates” is another occurrence of that symbol somewhere. Although in a narrow sense tokens are objects which are thus being created, the term “construction” emphasizes the distinction between this behavior and general object creation, which is defined in terms of handle validation.

String concatenation similarly doesn’t really do much creation. Executing ab//cd constructs another occurrence of abcd somewhere, without creating any new symbols.

The same applies to extensional sets and lists. The set {1,2} has always existed and always will, and there’s only one of it, regardless of how many times we write {1,2}. It seems to make sense that we regard {1,2} as a constructor expression yielding a token denoting that one eternally existing set, just as 2+2 yields a token denoting one eternally existing number. The form of the token corresponding to {1,2} is entirely a matter of implementation.

7 IMPLEMENTATION FACTORS

Practical implementations limit the lengths of handles, inducing compromises much as fixed word-length machines compromise pure arithmetic with truncation and roundoff errors. This has far-reaching consequences in the treatment of large data values, and in the confusion between extensional and intensional sets and lists. It induces differences in the treatments of things based on the size of their representations.

7.1 A Naive Implementation

In a naive implementation, with no length restriction and no synonymous handles, tokens referring to numbers and strings would directly contain their representations. Tokens referring to created objects would be copies of their assigned handles.

Whatever the data model or storage organization, particular tokens are involved in particular pieces of information. If two books are each 100 pages long, distinct tokens referring to 100 would be associated with each book. If two books were written by the same author, distinct tokens referring to the author would be associated with each book. If two books contained the same diagram, distinct tokens referring to the diagram would be associated with each book.

This provides the following semantics:

- Whether the book has the same length, or the same author, or contains the same figure can be determined directly by comparing tokens.
- The length, author, or included figures of one book could be altered independently of the other. The alteration would consist of replacing one token with another. That's the net effect, even if the process consisted of something like arithmetic or string manipulation.

Tokens referring to extensional sets and lists could directly contain references to their members. A book having several authors might have an associated token of the form {@smith, @jones}; for sets, the ordering would be canonical to facilitate comparison. Like references to numbers, the identity of the set could be determined by syntactic analysis of the handle. Each book would have its own authors token. The semantics would be as before:

- Whether two books have the same authors can be determined by direct comparison of the authors token. Being extensional sets, a match would mean that the authors of each book are one and the same set.
- The authors of one book could be altered without affecting the other. Its authors token would be replaced by a different token. (The actual operation may again take the form of string manipulation.)

This is essentially what is meant by copy semantics. If contracts are individually negotiated for each book, then each book would have its own contract token containing the text of the contract, i.e., its own copy. Two books would have the same contract if and only if these tokens matched. The contract for one book could be modified without affecting another. Assigning a contract from one book to another implies copying the token, leaving them subsequently independent.

In contrast, sharing semantics could be provided by creating carrier objects. If book contracts were selected from a set of standard contracts, they would be represented as carrier objects. The contract token associated with each book would refer to a carrier object. The text of the contract would be in the cargo token associated with the carrier object. By definition, books whose contract tokens refer to different carriers have different contracts — even if the associated text happens to be the same (that's where deep equality comes in). Altering the text of a contract alters the contract for all books referring to the corresponding carrier. Assigning a contract from one book to another copies the token referring to the carrier. Changes to the associated text apply to both books.

The same could be done for sets and lists. If each book in a series had the same set of authors, an intensional author set would be created as a carrier object. Each book in a series would have an authors token referring to the same carrier. Books whose authors tokens refer to different carriers refer to different authors objects, even if the associated sets are the same. Altering the set of authors in the cargo of a carrier alters the authors of all books referring to that carrier. And so on.

7.2 The Effect of Limited Handle Lengths

Direct tokens are like the ones just described, directly containing representations of numbers, strings, sets, and lists.

Limited handle lengths in practical implementations constrain the numbers and strings that can be represented in direct tokens, and generally precludes the use of direct tokens for sets and lists. *Indirect tokens* are used instead, containing pointers to other parts of storage containing the actual representation of long strings, sets, and lists. (Very large numbers don't seem to occur often enough to warrant this treatment.)

The remote data is logically an extension of the token. However, certain implementation idiosyncrasies alter the semantics, often to the point where it seems that carrier objects are being implicitly introduced, confusing the distinction between copy and sharing semantics.

The key concern arises when the remote data for two indirect tokens is the same. If the remote data is stored separately for each token, the pointers will be different. Imprecise implementations which only compare the pointers instead of the remote data will treat the equal tokens as unequal. The pointers will take on the status of carrier objects. Extensional sets become intensional sets, being different even if they have the same members. Precise implementations, in contrast, will compare the remote data instead of the pointers.

Alternatively, an implementation might avoid redundancy in storage by letting matching tokens point to the same remote data. In this case, imprecise implementations will modify tokens in place, inadvertently altering all indirect tokens sharing that data. The pointers again take on the status of carrier objects. Precise implementations will only modify tokens in place if no other indirect tokens are sharing the remote data. Shared remote data is more complicated, requiring much searching to avoid duplicate storage, as well as keeping track of whether and how many indirect tokens are currently pointing to a remote data item.

User intentions with respect to copy and sharing semantics can be expressed by the presence or absence of explicitly created carrier objects. Precise implementations respect these semantics. Imprecise implementations impose their own interpretations based on the size of the representation of data and various paradigms for efficient management of the storage resource.

8 WHAT'S AN OBJECT?

Are numbers objects? Are sets? Of course, it all depends on what you mean by "object". We've avoided the question as long as we can, and won't even really deal with it now.

Some models assume that objects are things inside the system, others allow them to be entities outside. In order to apply our identity concepts to models which assume objects are inside the system, there must be some mapping of our notion of handle to their notion of "where" an object is. We don't know what that mapping is.

In many models, the term "object" is reserved for explicitly created things, while the eternally existing ones might be called values, or literals, or something similar. We sidestep that issue by simply exploring similarities and differences, leaving the final decision on terminology to arbiters. But this paper undoubtedly reveals the author's bias toward considering them all to be objects.

Eternal objects and created objects are similar in that they are all referenced by handles. They are different in that eternal objects aren't explicitly created, and their handles are recognizable by syntactic means.

Sets and similar aggregates appear to fall in the middle ground, exhibiting both kinds of behavior at times. That's an illusion, neglecting the distinction between extensional and intensional sets. Extensional sets behave much like numbers. Intensional sets are created objects, i.e., carriers having extensional sets as their cargos. Extensional sets are implemented differently from numbers because of length limitations on handles, but their semantics have much in common. They always exist so long as their members exist, and references to them could be recognized syntactically.

The carrier and cargo concepts resolve some ambiguities of identity for aggregate objects. If you remove something from a stack, is it still the same stack? Yes and no. The stack carrier is still the same object; its cargo, a particular list of things, has been replaced by a different list.

What does it mean to insert the same tuple into two relations? What does a "tuple identifier" (tid) identify? How can a relation admit duplicate tuples if they are identified by unique tids? Is a tuple still the same tuple after it is updated? Again, yes and no. Insertions and deletions in a relation insert and delete tuple carrier objects; they are identified by distinct tids. The cargo of such a carrier is a tuple value. Inserting the "same tuple" into two relations means that two tuple carriers have the same value as their cargo. Duplicates tuples in a relation also means that two carrier objects are carrying the same cargo. After updating a tuple, the carrier object is still the same, but its cargo is different.

Whoever said "You never step in the same river twice" was making this distinction. The river as a geographic entity is a persistent carrier object. The river as a particular configuration of specific water molecules is its cargo, continuously changing. The cargo is never the same, though the carrier persists.

9 CONCLUSIONS

Let's summarize the key points.

Objects constitute an arbitrary set of things, which may or may not be in the computational system.

Symbols constitute a fixed set of distinguishable abstractions outside the computational system. Tokens constitute a varying set of occurrences of symbols; we are interested in those occurring within the computational system. Each token is an occurrence of exactly one symbol.

A scope is an arbitrary object such that each token belongs to exactly one scope. An operation in the computational system is executed within a scope.

A handle is a symbol which uniquely refers to one object in a given scope. Such handles are valid in the scope. These notions are formalized in the conceptual mappings

$$\textit{Valid: Symbol} \times \textit{Scope} \rightarrow \textit{Boolean,}$$

$$\textit{Referent: Symbol} \times \textit{Scope} \rightarrow \textit{Thing.}$$

The mappings are conceptual in the sense that they are not executable within the computational system.

All occurrences of a symbol in a given scope refer to the same thing. The validity and referent of a token in the system can also be conceptually defined in terms of its corresponding symbol:

$$\textit{Sym: Token} \rightarrow \textit{Symbol,}$$

$$\textit{Sc: Token} \rightarrow \textit{Scope,}$$

$$\textit{Valid(token)} ::= \textit{Valid(Sym(token), Sc(token)),}$$

$$\textit{Referent(token)} ::= \textit{Referent(Sym(token), Sc(token)).}$$

The computational system knows of the existence and identity of objects in terms of the validity of tokens as references within a scope. These can be modeled in terms of *Valid* and *Identity* predicates, which are executable operations in the computational system. (They may or may not be provided in an actual implementation.)

If *Valid(t)* is true for a token *t*, it means that *t* is a reference, i.e., an occurrence of a valid handle in the scope of *t*. The validity of some handles, such as references to mathematical abstractions, is recognized syntactically, i.e., their validity is computed by a parsing procedure. Validity of other handles is established by creation operations, which render previously invalid symbols valid. Like all operations, a creation operation is executed in a given scope; it introduces awareness of an object into that scope.

Computational operations such as arithmetic, string manipulation, and set specification may introduce occurrences of symbols into a system, but they do not create symbols nor the things denoted by symbols.

The *Identity* predicate $t_1 \equiv t_2$ is true in a scope if the tokens t_1 and t_2 are valid references to the same thing within that scope. In a simple system without synonymous handles, identity corresponds to equality of reference tokens. With synonymous handles, more elaborate algorithms are required.

If $t_1 \equiv t_2$ is true in a scope:

- t_1 and t_2 must both be valid handles in the scope.
- t_1 and t_2 should not have been rendered valid by distinct creation events, unless they have been explicitly coerced to denote the same object.
- If there is an extensional object set constructor {}, then the cardinality of $\{t_1,t_2\}$ must be 1.

If t_1 and t_2 denote the same number, extensional set, or other such abstraction, then $t_1 \equiv t_2$.

An operator *f* is *object-based* if replacing t_1 with t_2 in $f(t_1)$ would not change the result whenever $t_1 \equiv t_2$. Otherwise *f* is *valued-based*.

The distinction between intensional and extensional objects is modeled by carrier objects and their cargos.

The judgement as to whether or not certain things are the same object must be made outside the computational system. Such decisions are then reflected in the assignment of handles, the behavior of an *Identity* predicate (if implemented), and in the behavior of object-based operators.

10 ACKNOWLEDGMENTS

This paper has benefitted from some penetrating discussions with Alan Snyder and Walt Hill at H-P Labs, though it does not reflect consensus. Valuable comments have also been provided by Rafi Ahmed, Peter Lyngbaek, and Ming Shan.

11 REFERENCES

[Atkin89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik, “The Object-Oriented Database System Manifesto”, Proc First Intl Conference on Deductive and Object-Oriented Databases, Dec., 1989, Kyoto, Japan.

[Hall76] P.A.V. Hall, J. Owlett and S.J.P. Todd, “Relations and Entities”, in *Modeling in Data Base Management Systems*, G.M. Nijssen, ed., North-Holland, 1976.

[Khosh86] Setrag Khoshafian and George Copeland, “Object Identity”, Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Portland, Oregon, 1986. Also in [ZM1].

[Zdoni89] Stanley Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*, Morgan Kaufmann, San Mateo, California, 1989.