Oaklisp: an Object-Oriented Scheme with First Class Types

Kevin J. Lang and Barak A. Pearlmutter

Department of Computer Science

Carnegie-Mellon University

Pittsburgh, PA 15213

Abstract

The Scheme papers demonstrated that lisp could be made simpler and more expressive by elevating functions to the level of first class objects. Oaklisp shows that a message based language can derive similar benefits from having first class types.

Introduction

Oaklisp is a message based, multiple inheritence dialect of lisp. Programs are written using lisp syntax, and traditional lisp data types coexist with a Smalltalk style class hierarchy. This paper assumes that the reader is familiar with one of the many object-oriented lisp dialects of this sort, and will therefore concentrate on the unique aspects of Oaklisp which are mostly due to the influence of Scheme.

Oaklisp is based on Scheme in two ways. Scheme was used as the model for syntactic details whenever possible in order to minimize our contribution to the continual proliferation of incompatible varieties of lisp. More significantly, Oaklisp is based on the Scheme philosophy, which states that the primitive forms of a language should be simple, powerful, and meaningful from several points of view. The careful design of Oaklisp permits its object-oriented and procedural sides to be more closely integrated than in a language which just hangs a separate message facility on the side of an existing lisp. Although Oaklisp is object-oriented from the core, all of its features behave in such a way that pure Scheme emerges as an alternate programming style.

Because Oaklisp is so closely related to Scheme, it is worth taking a look at the main ideas of Scheme before proceeding. The conceptual foundation of the language is that functions are objects just like everything else, which means they can be returned from calls, passed around, stored in data structures, and so forth. This principle has several implications that are not immediately obvious. Because a function can be applied at a point distant in time and space from its point of origin, it must be able to remember the bindings of any variables that were visible when it was made. This additional complexity is offset by the ability to write many previously primitive control structures at the user level, and by the fact that the special mechanisms that lisp ordinarily uses for defining and applying functions can be dispensed with.

In lisp, the car position of a function call is treated as the name of a function which is looked up in a special table and then applied to the values obtained by evaluating the arguments of the call. In Scheme, the car of a call is an evaluated position. Although any expression can occur in the car, it is common for the expression to be a variable, in which case a call looks exactly like it would in lisp even though something completely different is going on. For example, the Scheme form (PLUS 1 2) is evaluated by looking up the binding of the variable PLUS and applying the resulting function to the values 1 and 2. Because functions are manipulated using the same mechanisms as other forms of data, they are *first class*. Because functions are never found by looking up their name, they are *anonymous*. It is worth pointing out that when a function is commonly bound to a particular variable (such as PLUS), it is convenient to speak as if the variable's name were the function's name. This practice should not be allowed to obscure the fact that the function is really an anonymous object which happens to be accessible through a standard variable binding.

The Oaklisp version of a function call is an amplified version of what we have just seen in Scheme, with identical syntax and closely related semantics. The first step in the evaluation of a call is the same, namely the recursive evaluation of the subelements of the form. The message based semantics of Oaklisp only becomes manifest in the application step of evaluation, where the car value is taken to be an operation and the second value is taken to be an object whose type determines the method which is invoked to perform the operation. The remaining arguments are passed along to the method, but play no role in its selection. It should be clear that this message passing paradigm is basically the same as in Smalltalk. The inheritance and shadowing of methods occurs in the usual way. The only major difference between Oaklisp and Smalltalk messages is that Oaklisp operations are not

[1]This work was supported by grants from DARPA and the System Development Foundation. Barak Pearlmutter is a Hertz Fellow.

symbols; they are anonymous objects that may be passed around and compared. The anonymity of operations is necessary so that Oaklisp will have the correct Scheme semantics when a functional programming style is adopted. However, an operation is not a function by itself, since it is not directly associated with any code. An operation is just a thing with a unique identity that in conjunction with a type specifies a method which can actually be executed.

To make all of this a bit more concrete, consider the evaluation of the call (PLUS 2 3). The first subform is a variable which is dereferenced, yielding an operation. The other two subforms are constants, so they evaluate to themselves. The type of the distinguished first argument is retrieved, and then the method tables of the integer type and its supertypes are searched using the anonymous operation as a key. If the tables have been set up correctly, the search leads to the selection of a method that knows how to add things to an integer. Finally, the method is invoked with the arguments 2 and 3.

We have seen how the evaluation of calls is moderated by the type system, and in particular by the method tables of types. The ability to configure these tables is provided by the ADD-METHOD special form. For example, by evaluating the following expression we can define a PLUS method for cons cells such that (PLUS (CONS 2 3)) → 6.

```
(ADD-METHOD (PLUS (PAIR) SELF)
  (TIMES (CAR SELF) (CDR SELF)))
```

This form tells the system to associate the method specified by the body with the operation PLUS in the method table of the type PAIR. It is a special form rather than a call because of the keyword ADD-METHOD and because the body and argument list (SELF) are not evaluated. However, the operation and type positions *are* evaluated, which means that PLUS is just a variable which is bound to the same operation which will later be used as a method selector during a call.

We have repeatedly said that operations are things that can be passed around and stored in variables. The reader may wonder where operations come from in the first place. Since they are objects just like everything else, they are obtained the same way as any other object, namely by instantiating a type. An instance of a type is generated by sending the type a MAKE message. For example, the system PLUS operation is defined by evaluating

```
(SET! PLUS (MAKE OPERATION)).
```

Now according to the Oaklisp evaluation rules for calls, OPERATION is not the name of a type but is a variable bound to a type object. Since types are anonymous things just like operations, they are obtained in an analogous manner, by instantiating the type TYPE. For example, we could create a frog type with the following expression.

```
(SET! FROG (MAKE TYPE (LIST 'AGE 'COLOR) (LIST OBJECT)))
```

In this case the MAKE message takes extra arguments which specify the new type's instance variables and supertypes. Because the MAKE expression is a call and not a special form, all of its subexpressions are evaluated. Observe that the first argument expression has been written in a manner that evaluates to a list of symbols, and that the second yields a list of type objects.

The type specified as the supertype of FROG is OBJECT, which is the distinguished type that lives at the top of the inheritance graph. Because every type is a subtype of OBJECT, a method which has been installed in OBJECT will work on any object whatsoever and can be used as the default method for an operation. This ability to define default methods suggests the following strategy for setting up a type predicate.

```
(SET! FROG? (MAKE OPERATION))

(ADD-METHOD (FROG? (OBJECT) SELF)
  NIL)

(ADD-METHOD (FROG? (FROG) SELF)
  T)

(SET! FRED (MAKE FROG))

(FROG? FRED)  →  #!TRUE

(FROG? FROG)  →  ()
```

The last two expressions illustrate the fact that while FRED is a frog, FROG is a type. FRED and FROG are also objects, because FROG and TYPE are subtypes of the type OBJECT. These and other amazing facts can be seen in figure 1, which shows the primordial type hierarchy together with the frog example. It is important to understand the difference between the *is-a* and *subtype-of* relations. Whereas every object is the bottom of an *is-a* link, types are the only participants in *subtype-of* links.

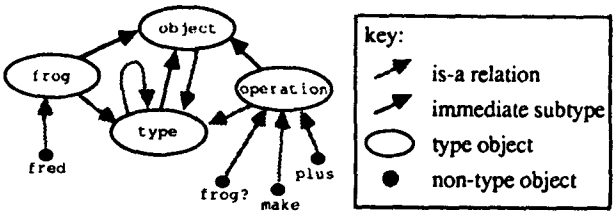

Figure 1: The Primordial Types

Although the primordial types OBJECT, TYPE, and OPERATION may seem to be magic, they are just like any other types and could be defined at the user level.[2] The general lack of magic in the type system permits a degree of openness and extensibility beyond that of Smalltalk and Zetalisp, both of which have lots of special machinery laying around. The later sections of this paper present several nifty features that can be defined in Oaklisp at the user level.

But first, there are a couple of details that need to be addressed in the frog example. The FROG type possesses instance variables which should be initialized when a frog is made. Since this is a common requirement, the MAKE method for types sends new instances an

[2] Actually, some pointer tweaking is needed because of the circularities in their definitions.

INITIALIZE message before returning them. The default INITIALIZE method is a no-op. By specifying an INITIALIZE method for frogs, we can shadow out this default method and cause something useful to happen.

```
(ADD-METHOD (INITIALIZE (FROG AGE COLOR)
                        SELF INITIAL-COLOR)
   (SET! AGE 0)
   (SET! COLOR INITIAL-COLOR))
(SET! FRANK (MAKE FROG 'OLIVE-DRAB))
```

Notice that when an instance variable is used in the body of a method, it must be declared at the top of the method. This helps to disambiguate variable references for both the compiler and the programmer.

A particularly important aspect of the ADD-METHOD form is that the method is closed in its lexical environment when the form is evaluated. Together with the fact that ADD-METHOD returns its operation argument, this rule allows the LAMBDA special form of Scheme to be defined with the following macro.

```
(LAMBDA arg-list. body) ≡
  (ADD-METHOD ((MAKE OPERATION) (OBJECT). arg-list). body)
```

When a LAMBDA form is evaluated, it generates a new anonymous operation and supplies a default method for the operation. Since the same piece of code is invoked every time the operation is sent in a message, the operation behaves exactly like a function. Using the LAMBDA form, Oaklisp programs can be written in a functional style that is indistinguishable from Scheme. In practice, programs tend to be written in a mixture of the functional and object-oriented styles. It is easy to combine the two styles in a harmonious manner because the Scheme component of the language is just the natural result of having anonymous operations and a lexically scoped ADD-METHOD form.

The Oaklisp Cons Hierarchy

In the Oaklisp kernel, conses are defined in Oaklisp itself in a way open to extension by ordinary users. The cons hierarchy (see figure 2) is rather detailed, allowing each method and subtype to be defined at the right level of abstraction.

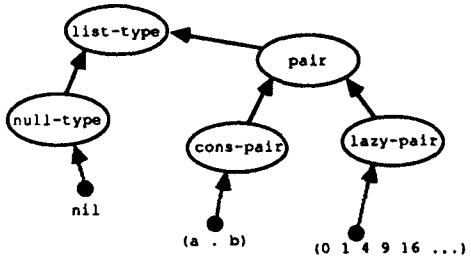

Figure 2: The Cons Hierarchy

```
(SET! LIST-TYPE (MAKE TYPE '() '()))
(SET! NULL-TYPE (MAKE TYPE '() (LIST LIST-TYPE OBJECT)))
(SET! PAIR (MAKE TYPE '() (LIST LIST-TYPE)))
(SET! CONS-PAIR (MAKE TYPE '(THE-CAR THE-CDR) (LIST PAIR OBJECT)))
(ADD-METHOD (CAR (CONS-PAIR THE-CAR) SELF)
   THE-CAR)
```

The PAIR type is never instantiated; it is an abstract type for "things that behave like lisp conses." Methods for printing and mapping are defined at the PAIR level and are shared by all of PAIR's subtypes, while the subtypes themselves are responsible for handling CAR and CDR messages. Ordinary cons cells are instances of the type CONS-PAIR, but other useful subtypes of PAIR can be defined as well. For example, the following program fragment sets up a type of lazy pair that only computes its car and cdr when they are actually needed.

```
(SET! LAZY-PAIR
      (MAKE TYPE '(CAR-THUNK CAR-FLAG CDR-THUNK CDR-FLAG)
                 (LIST PAIR OBJECT)))
(ADD-METHOD (CAR (LAZY-PAIR CAR-THUNK CAR-FLAG) SELF)
   (COND (CAR-FLAG CAR-THUNK)
         (ELSE (SET! CAR-THUNK (CAR-THUNK 'IGNORE))
               (SET! CAR-FLAG T)
               CAR-THUNK)))
(ADD-METHOD (INITIALIZE (LAZY-PAIR CAR-THUNK CAR-FLAG
                                   CDR-THUNK CDR-FLAG)
                        SELF ORIGINAL-CAR-THUNK ORIGINAL-CDR-THUNK)
   (SET! CAR-FLAG NIL)
   (SET! CAR-THUNK ORIGINAL-CAR-THUNK)
   (SET! CDR-FLAG NIL)
   (SET! CDR-THUNK ORIGINAL-CDR-THUNK))
```

When we make a lazy-pair, we give it "thunks" for the car and cdr values. The pair then uses a call by need strategy, in which a thunk is used to compute the car or cdr on first request, and the computed value is stored and returned immediately on future requests. The ease with which we can create thunks is a consequence of the Scheme scoping rules, which allow us to close a function in the environment of its creation and to use the function even after the environment in which it was created has been exited. When we want to make a thunk for a computation, we just close a function to compute the needed value in the appropriate environment. For instance, to create an infinite list of squares we can write

```
(SET! MAKE-OMEGA-SQUARES-FROM
   (LAMBDA (N)
      (MAKE LAZY-PAIR (LAMBDA (IGNORED) (* N N))
                      (LAMBDA (IGNORED)
                         (MAKE-OMEGA-SQUARES-FROM (+ N 1))))))
(SET! INFINITE-SQUARELIST (MAKE-OMEGA-SQUARES-FROM 0)).
```

The syntax here is somewhat awkward[3] but syntax is not the point of the example. We have created an infinite list which is computed on demand. Since we've built on the abstract PAIR type, the list can be printed and manipulated like any other list; our lazy pairs deal with CAR and CDR a little ideosyncratically, but that's invisible from outside the type. For instance, if we were to now print INFINITE-SQUARELIST, "(0 1 4 9 16 25 36 49 ...)" would appear on our screen. It is interesting to note

[3]Expert lisp programmers will recogize the opportunity to define a LAZY-CONS macro.

that the printer normally abreviates long lists, printing "..." after a certain number of elements have been printed out. This feature is inherited by all subtypes of pair, so we don't have to do anything special to make lazy pairs print reasonably. This illustrates the usefulness of abstract types and the importance of separating them from particular implementations. It also shows the usefulness of defining very general methods at high levels of abstraction, as such a policy leads to greater code sharing.

It is equally easy to make a list that is overlayed onto a fractal set of points on the screen or a string that is the mapped image of a file. These examples illustrate the synergy between the dual linguistic paradigms embodied in Oaklisp. The object orientation of Oaklisp allows us to write modular definitions that can be hooked right into the type hierarchy, providing the user with a palette of types having uniform behavior but varying implementations. The Scheme semantics of Oaklisp allows these implementations to play games with higher order functions, using deviant Conniver-style control structures of the sort that provided the original motivation for developing Scheme and Prolog.

Coercable Types

In this section we define a new metatype, COERCABLE-TYPE, whose instances are types that possess coercion operations. In order to coerce an object to one of these types, we send a message to the type asking it to return its coercion operation, and then we apply the resulting operation to the object. For example, we might want to define a new kind of table that associates keys and values like a hash table but is implemented using a self-adjusting binary tree. By evaluating the expression

```
(SET! SLEATOR-TABLE
      (MAKE COERCABLE-TYPE '(ROOT SIZE) (LIST TABLE OBJECT)))
```

we can create the type and have a coercion operation generated automatically. To define methods for this operation, we can get our hands on it by sending the sleator-table type a COERCER message. Then, if FOO is bound to a table of some other sort, we can coerce it to a sleator-table by evaluating

```
((COERCER SLEATOR-TABLE) FOO).
```

Now that we have seen how coercable types should behave, we can actually define them. Observe how the anonymity of operations and types permits us to use type objects to govern access to coercion operations. First we create the new metatype with

```
(SET! COERCABLE-TYPE (MAKE TYPE '(COERCION-OP) (LIST TYPE))).
```

This new metatype is just like the original metatype TYPE except for a new instance variable COERCION-OP. Next we define the interface to coercable types: the COERCER operation.

```
(SET! COERCER (MAKE OPERATION))

(ADD-METHOD (COERCER (COERCABLE-TYPE COERCION-OP) SELF)
   COERCION-OP)
```

It remains to define an INITIALIZE method for coercable types so that when one is created it will make itself a coercing operation and stash it in its COERCION-OP instance variable.

```
(ADD-METHOD (INITIALIZE (COERCABLE-TYPE COERCION-OP)
                        SELF IVARS SUPERTYPES)
   (CONTINUE SELF IVARS SUPERTYPES)      ;Do inherited initializations
   (SET! COERCION-OP (MAKE OPERATION))   ;Make the coercion operation
   (ADD-METHOD (COERCION-OP (SELF) INSTANCE-OF-SELF)
      INSTANCE-OF-SELF))                 ;How to coerce instances of ourself
```

The expression (SET! COERCION-OP (MAKE OPERATION)) is straightforward, but the other two forms in the initialization code require some explanation. The first form, (CONTINUE ...), runs handlers for the INITIALIZE operation that lie above the current point in the type hierarchy. The code following the (CONTINUE ...) is executed after these handlers, much like a ZetaLisp :AFTER method. The expression (ADD-METHOD (COERCER (SELF) ...) ...) adds a handler to the type that is being created. The handler is for the type's coercion operation, so this form tells instances of the type how to be coerced to that type. Since they are already of the right type, all they need do is return themselves.

To give a concrete example of how this facility is used, imagine that we want to add complex numbers to our language, and that we want our complex numbers to have two different representations, cartesian and polar. We'd also like to be able to switch between representations conveniently. Our implementation will use an abstract type COMPLEX and two concrete subtypes.

```
(SET! COMPLEX (MAKE TYPE '() (LIST NUMBER)))
(SET! ORTHO-COMPLEX (MAKE COERCABLE-TYPE '(REAL-COMPONENT
                                           IMAG-COMPONENT)
                                         (LIST COMPLEX OBJECT)))
(SET! POLAR-COMPLEX (MAKE COERCABLE-TYPE '(ANGLE LENGTH)
                                         (LIST COMPLEX OBJECT)))
```

Now that we have some new types of numbers, we have to make them do all the things numbers are supposed to: addition, exponentiation, printing themselves, etc. The details are tedious; for expository purposes, a few examples suffice.

```
;; Absolute value: expensive in one representation.
(ADD-METHOD (ABS (ORTHO-COMPLEX REAL-COMPONENT IMAG-COMPONENT)
                 SELF)
   (SQRT (+ (EXPT REAL-COMPONENT 2) (EXPT IMAG-COMPONENT 2))))

;; but cheap in the other.
(ADD-METHOD (ABS (POLAR-COMPLEX LENGTH) SELF)
   LENGTH)

;; Getting the imaginary component:
(SET! IMAGPART (MAKE OPERATION))


;; Cheap in one representation.
(ADD-METHOD (IMAGPART (ORTHO-COMPLEX IMAG-COMPONENT) SELF)
   IMAG-COMPONENT)

;; but expensive in the other.
(ADD-METHOD (IMAGPART (POLAR-COMPLEX ANGLE LENGTH) SELF)
   (* LENGTH (COS ANGLE)))

;; and trivial for non-complex numbers.
(ADD-METHOD (IMAGPART (NUMBER) SELF)
   0)

;; Raising to a power.
(ADD-METHOD (EXPT (POLAR-COMPLEX ANGLE LENGTH) SELF POWER)
   (MAKE POLAR-COMPLEX (* ANGLE POWER) (EXPT LENGTH POWER)))
```

```
;; We define equality at an abstract level so that both representations will
;; inherit how to compare for equality.
(ADD-METHOD (= (COMPLEX) X Y)
   (AND (= (REALPART X) (REALPART Y))
        (= (IMAGPART X) (IMAGPART Y))))
```

Now we get to use our powerful coercion operations. Each kind of complex number automatically knows how to coerce to itself; we just have to tell each kind how to be coerced to the other.

```
;; What a polar does when it receives an ortho coercion message:
(ADD-METHOD ((COERCER ORTHO-COMPLEX) (POLAR-COMPLEX) SELF)
   (MAKE ORTHO-COMPLEX (REALPART SELF) (IMAGPART SELF)))

;; What an ortho does when it receives a polar coercion message:
(ADD-METHOD ((COERCER POLAR-COMPLEX) (ORTHO-COMPLEX) SELF)
   (MAKE POLAR-COMPLEX (PHASE SELF) (ABS SELF)))
```

With everything defined, we're ready to demonstrate. Lets create a complex number represented in cartesian terms and coerce it into one represented in polar terms.

```
;; First we create sqrt(-1), representing it in cartesian terms.
(SET! I-ORTHO (MAKE ORTHO-COMPLEX 0.0 1.0))
  →  #CO(0.0 1.0)

;; We can verify that it works.
(* I-ORTHO I-ORTHO)
  →  -1.0

;; Now we coerce it to another representation.
(SET! I-POLAR ((COERCER POLAR-COMPLEX) I-ORTHO))
  →  #CP(1.0 1.5707963267949)

;; And verify that the alternate representation functions correctly.
(* I-POLAR I-POLAR)
  →  -1.0

(* I-ORTHO I-POLAR)
  →  -1.0

;; Note that the two representations are different objects...
(EQ? I-ORTHO I-POLAR)
  →  ()

;; but are numerically equal.
(= I-ORTHO I-POLAR)
  →  #ITRUE
```

An interesting consequence of this coercable type mechanism is that we can try to coerce one thing to the type of something else without knowing what that second type is. For example, suppose FOO is a hash table and BAR is some kind of table that seems very fast and efficient, and we'd like to coerce FOO into a table of the same sort. We get BAR's type, ask it for its coercion operation, and apply that operation to FOO, as follows:

```
((COERCER (GET-TYPE BAR)) FOO).
```

The contrast between our coercable type construction and Smalltalk class variables is also interesting. In Smalltalk, there are special variables which are global to an entire type. The COERCION-OP instance variable of each coercable type is morally equivalent to a class variable, since any instance of a coercable type could get to the coercable operation by running up its *is-a* link. By defining an interface at the type level allowing the variable to be accessed and some macros to sugar the syntax, we could use this sort of definition to make things that look almost exactly like class variables. All of this activity can take place at user level—no modification or knowledge of system internals is necessary.[4]

Mixin Managers

Frequently, type hierarchies become so rich that they threaten to overwhelm users with a plethora of possible combinations of mixins. The combinatorial explosion of the number of possible concocted types seems intrinsic to the style of programming involving multiple functionally orthogonal mixins. Above a certain level of complexity, finding a type with certain known characteristics can become difficult. Programmers are left wondering "Has a type based on *foo* with *bar*, *baz* and *zonk* mixed in been created, if so what's its name, and if not what should I name it and where should I define it?"

In Oaklisp, it is easy to define mixin managers that take care of this problem. When programmers need "the type based on *foo* with *bar*, *baz* and *zonk* mixed in," they ask a mixin manager for it. If such a type has already been created, it is returned; if not, the mixin manager creates an appropriate new type, caches it, and returns it. This relieves programmers of the burden of remembering which types have been concocted and what they are named.

It is enlightening to examine the :MIXTURE defflavor option added to the flavors system in Symbolics ZetaLisp Release 5. Although :MIXTURE provides functionality vaguely similar to that of an Oaklisp mixin manager, the implementation of :MIXTURE required major additions to the deeply internal definitions of DEFFLAVOR and other portions of the flavor system, and was far from being a user level extension. The reason :MIXTURE was so difficult to define is that ZetaLisp flavors are not first class. In Oaklisp, on the other hand, mixin managers are defined at user level.

```
(SET! MIXIN-MANAGER (MAKE TYPE '(CACHE) (LIST OBJECT)))

(SET! MIX (MAKE OPERATION))

(ADD-METHOD (MIX (MIXIN-MANAGER CACHE) SELF TYPE-LIST)
   (LET ((X (ASS EQUAL? TYPE-LIST CACHE)))
      (COND (X (CDR X))   ;found type in cache
            (ELSE
             ;;Not found: create a new type and stash it.
             (LET ((NEW-TYPE (MAKE TYPE '() TYPE-LIST)))
                (SET! CACHE (CONS (CONS TYPE-LIST NEW-TYPE)
                                  CACHE))
                NEW-TYPE)))))
```

When an instance of the MIXIN-MANAGER type receives a MIX message it gets one argument: a list of types to be mixed together. The mixin manager checks its cache, creating and caching the requested type if necessary.

To demonstrate a mixin manager in action, consider the Oaklisp operation hierarchy, which is quite elaborate. Some of the types and mixins involved are OPERATION, OPEN-CODABLE-OPERATION, SETTABLE-OPERATION, LOCATABLE-OPERATION, TAGTRAPABLE-MIXIN, and CONSTANT-FOLDABLE-MIXIN. Their precise functionality isn't relevent; what is of note is that when we make a new operation which should be a combination of a number of these types, we can use a mixin manager to

[4] The Scheme features of Oaklisp provide another way to allow a number of methods to share a variable of their own, even if the methods are for different types, but this works only if the method definitions share a single lexical scope.

help us. We proceed by first making a mixin manager and then using it to get a complex combination of types.

```
(SET! OPERATION-MIXIN-MANAGER (MAKE MIXIN-MANAGER))
(SET! + (MAKE (MIX OPERATION-MIXIN-MANAGER
                   (LIST TAGTRAPABLE-MIXIN
                         OPEN-CODABLE-MIXIN
                         CONSTANT-FOLDABLE-MIXIN
                         OPERATION))
              2 1 '((PLUS-2-STACK))))
```

This definition of + is actually drawn from the Oaklisp kernel, and is just like the definition of a regular operation, except for some extra mixins and initialization arguments. The mixins tell the compiler and runtime system special things about the operation, such as how to open code it in compiled code and what to do if the corresponding primitive instruction encounters a tag trap. The localization of all information about various aspects of addition in the addition operation itself is a great boon to modularity. Contrast this with the approach that is necessary in most systems, where information about addition is distributed between the operator itself, the compiler internals, and the runtime system. The clean interface to the runtime system and compiler in Oaklisp is made possible by the anonymity of the objects and the general type hierarchy.

Semantic Foundations

Types represent sets of objects. This relationship may be specified by a mapping m which sends a type to the set of objects that it represents. In Oaklisp, types are themselves objects, so m is actually a partial mapping from objects to sets of objects. Figure 3 is a Venn diagram that depicts the mapping m and its interaction with the subtype predicate defined by the type system and the actual subset relation among sets of objects. The right hand side of the figure illustrates multiple inheritance, and the COERCABLE-TYPE stuff shows how new metatypes fit into the system.

The mapping m may defined more formally as follows. Let O be the set of all objects, and T the set of all type objects. Let $m: T \subseteq O \rightarrow 2^O$ be defined by $x \in m(a)$ if and only if (IS-A? x a). Interestingly, $T = m(\text{TYPE})$ and hence $\text{TYPE} \in m(\text{TYPE})$ since (IS-A? TYPE TYPE). Such near circularities raise a concern: when dealing with a system some of whose elements are members of sets represented by other elements, Russell's paradox may result.

In order to allay concerns of this sort, it is necessary to prove that the type heirarchy in Oaklisp cannot give rise to contradictions. Such a proof requires an axiomatic formalization, so that tools of logical analysis may be brought to bear. A beneficial side effect of such a formalization is that it serves as a succinct and precise way to express the semantics of the type heirarchy. A bit of the formalization is therefore presented here. We assume that the user will not redefine the primitives used. We let a and b range over types while x ranges over all objects. The notation $a \leq b$ is used for (SUBTYPE? a b).

$x \in m(\text{OBJECT})$.

The relation $\leq$ is a partial ordering of T.

If $a \leq b$ then $m(a) \subseteq m(b)$.

(GET-TYPE x) $\in T$.

If $x \in m(a)$ then (GET-TYPE x) $\leq a$.

(MAKE a ...) $\in a$.

$a =$ (GET-TYPE (MAKE a ...)).

If a is an element of the list l then (MAKE TYPE l ...) $\leq a$.

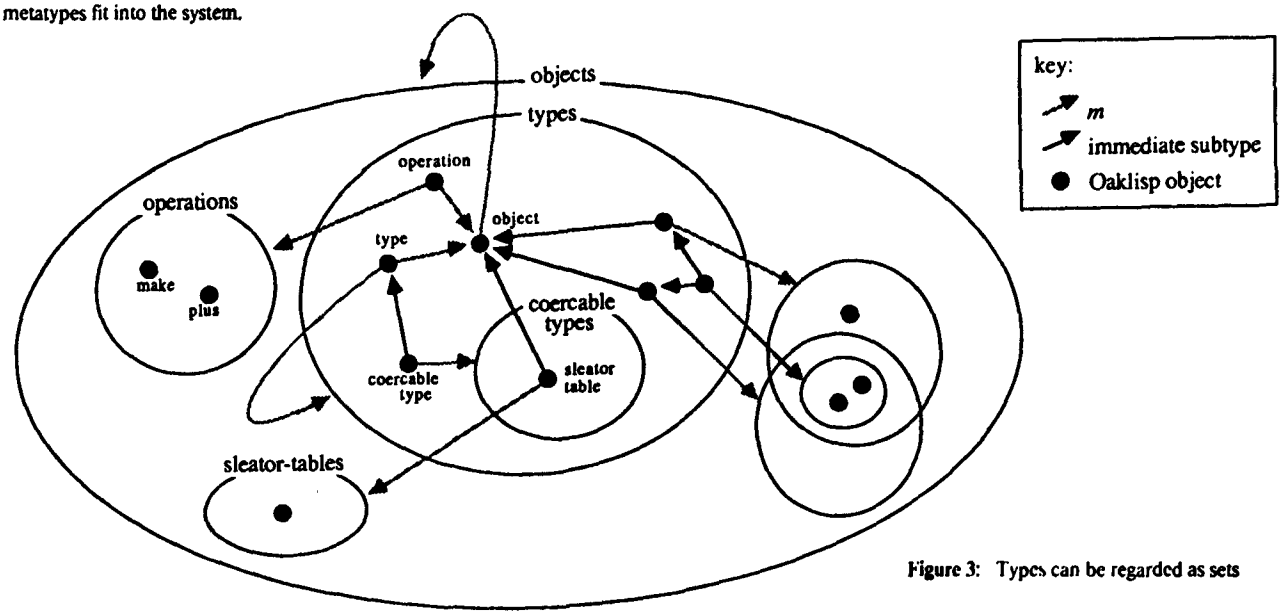

Figure 3: Types can be regarded as sets

It is noteworthy that the GET-TYPE function can be formalized in this way. The implementation of GET-TYPE returns the contents of the type field of an object. Formally, GET-TYPE returns the smallest type containing the object it is applied to. The guarantee that such a smallest nontrivial type exists for every object in the system is one way in which the theory of the Oaklisp type heirarchy differs from usual set theory, and was necessary for our proof of consistency.

Comparison to Other Work

Oaklisp derives its message sending syntax from T, a highly developed dialect of Scheme. However, T is not object-oriented in the usual sense since types are not visible to the user, there is no inheritance hierarchy, and it is impossible to add new methods to existing objects.

Object Lisp, Common Loops, and New Flavors also use the T syntax for messages. Oaklisp differs from these languages in that its type hierarchy lies at the heart of the language, eliminating the distinction between "regular lisp stuff" and "object-oriented extension stuff." More importantly, Oaklisp's types are first class and can be meaningfully manipulated by user code.

Conclusion

We stated at the beginning of the paper that the unique features of Oaklisp are mostly due to the influence of the Scheme philosophy. For example, the tight coupling between the object-oriented and functional sides of Oaklisp is motivated by the principle of not creating two primitive mechanisms when one will suffice. Scheme also supplied the idea of first class functions, whose power can be exploited even when the message based aspects of the language are being to used to maintain a conservative module discipline. The usefulness of first class functions inspired the first class types of Oaklisp, which turn out to have similar benefits, making it easy to define meta tools which deal with the semantic substrate of the language itself. In some ways, Oaklisp's ability to manipulate its own type structure is analogous to the ability of 3-lisp to reflect upon its own control structure.

Appendix: Current Implementation

The implementation of Oaklisp is heavily influenced by T, which was designed as a systems programming language. In particular, the Oaklisp compiler differs from those written for instructional dialects of Scheme by emphasizing run-time efficiency rather than hooks for the debugger. For instance, appropriate LABELS forms are compiled as tight loops that do not generate lambdas. This orientation also led us to provide for low-level access to native machine resources. Finally, T provided the inspiration for Oaklisp's lattice structured top level namespace, which is implemented with a collection of locale objects.

Oaklisp currently runs on the Macintosh™ personal computer[5]. It was cross-developed on a lisp machine, where a kernel interpreter was written in Common Lisp and the rest of the language was defined in terms of the kernel. A compiler was written in Oaklisp, targetting to a stack-oriented bytecode for which an emulator was written on the Macintosh. The current memory format uses two low tag bits, thus allowing 30-bit fixnums, pointers, and locatives that can address any word in the logical address space of the 68000. Characters, weak pointers, and Macintosh handles are represented by 24-bit immediate objects. There are facilities for so-called lightweight processes, and hooks into the window system and Macintosh toolbox, all written in Oaklisp. The current implementation effort has been driven by considerations of simplicity rather than speed. An Oaklisp implementation with efficiency as a primary goal could use all the usual tricks for speeding up lisps. For instance, making CONS-PAIR primitive would speed up list manipulation.

Bibliography

Harold Abelson et al.
The Revised Revised Report on Scheme
AI Memo 848, MIT Artificial Intelligence Laboratory, 1985.

D. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, F. Zdybel.
"CommonLoops: Merging Common Lisp and Object-Oriented Programming"
9th International Joint Conference on Artificial Intelligence,1985

W. Clocksin and C. Mellish.
Programming in Prolog
Springer-Verlag, 1981.

A. Goldberg and D. Robson.
Smalltalk-80: The Language and its Implementation
Addison-Wesley, 1983.

[5]Thanks to Bruce Horn for his help.

Sonya E. Keene and David A. Moon.
"Flavors: Object-oriented Programming on Symbolics Computers"
Common Lisp Conference, December 1985.

Glenn Krasner, Ed.
Smalltalk-80: Bits of History, Words of Advice
Addison-Wesley, 1983.

Kevin J. Lang and Barak A. Pearlmutter.
The Oaklisp Language Manual and *The Oaklisp Implementation Guide*
Unpublished, CMU Computer Science Department, 1985.

D. Moon and D. Weinreb.
The Lisp Machine Manual
Symbolics Inc.

Jonathan A. Rees and Norman I. Adams IV.
"T: a dialect of Lisp or, Lambda: the ultimate software tool"
Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, August 1982.

Jonathan A. Rees, Norman I. Adams IV, and James R. Meehan.
The T Manual
Fourth edition, Yale University Computer Science Department, 1984.

D. Sleator and R. Tarjan.
"Self-adjusting Binary Search Trees"
Proceedings of 15th Symposium on Theory of Computing, 1983.

Brian Cantwell Smith.
Reflection and Semantics in Lisp
CSLI-84-8, Center for the Study of Language and Information, 1984.

Guy Lewis Steele, Jr.
Lambda, the Ultimate Declarative
AI Memo 379, MIT Artificial Intelligence Laboratory, 1976.

Guy Lewis Steele, Jr. and Gerald Jay Sussman.
Lambda, the Ultimate Imperative
AI Memo 353, MIT Artificial Intelligence Laboratory, 1976.

Guy Lewis Steele, Jr. and Gerald Jay Sussman.
The Art of the Interpreter
AI Memo 453, MIT Artificial Intelligence Laboratory, 1978.

G. Sussman, and D. McDermott.
"From PLANNER to CONNIVER - A Genetic Approach"
Proceedings of the Fall Joint Computer Conference, 1972.
AFIPS Press.