8 The Implementation of Oaklisp

Barak A. Pearlmutter and Kevin J. Lang

8.1 Language Description

Oaklisp [4,5] is an object-oriented extension of Scheme; any program conforming to the R3RS Scheme standard [6] should run under Oaklisp with identical results. Thus, the most efficient way to describe Oaklisp is to mention the features that distinguish it from Scheme.

The fundamental computational model of Oaklisp is based on generic operations rather than functions. When an operation is applied to an object, the piece of code (or "method") that is actually invoked depends on the type of the object. For example, consider what happens when the expression `(car mylist)` is evaluated in Oaklisp. The variables `car` and `mylist` are dereferenced to yield two objects, the first of which is an operation – an anonymous token that denotes an abstract computation. This operation is then applied to the object to which the variable `mylist` was bound. If the object is an instance of the Oaklisp type `cons-pair`, [1] then a method is invoked which returns the contents of the first storage slot in the object. If the object is an instance of some type that doesn't have a `car` method, the run-time system checks to see whether a supertype of that type has a method for performing the operation. In case of conflict, the system invokes the method that would be found during a left-to-right, depth-first search of the multiple inheritance graph starting at the type of the object.

Syntactically, Oaklisp is similar to Scheme. Oaklisp has all the standard primitive special forms such as `if` and `quote`, but `lambda` is not primitive. In its place is the `add-method` special form, which associates a method with a type and an operation.

`(add-method (` *operation* `(` *type* `.` *ivarlist* `) .` *arglist* `)` *body*`)` *Special Form*

Returns *operation* after adding a method for *operation* to the method table of *type*. The arguments to the method are specified by *arglist*, and instance variables that are referenced in the method's body are declared in *ivarlist*. Free variables in a method are evaluated in the lexical context in which they appear, with the bindings from the time the form was evaluated, just as with `lambda` in regular Scheme. The *operation* and *type* positions of the form are evaluated at run time.

---

[1] The phrase "the type `cons-pair`" means "the type to which the variable `cons-pair` is bound." We will use this convention throughout this chapter.

The Oaklisp run-time environment includes all of the standard Lisp types such as numbers, conses, and symbols. Unlike other dialects of Lisp, types in Oaklisp are explicitly represented by objects that can be manipulated at run time. Thus, Oaklisp types are first-class entities in the same sense as Scheme functions. Oaklisp types are instantiated by the **make** operation. For example, the call **(make hash-table)** causes a message containing the **make** operation to be sent to the **hash-table** type object, which then allocates, initializes, and returns a new hash table object. Similarly, one can create a new operation by applying **make** to the **operation** type, and one can create a new type by applying **make** to the **type** type.

To see how Oaklisp works in practice, let us define a new type of cons cell using **make**.

```
(set! mycons-cell
      (make type
            (list pair)                    ;supertypes
            (list 'slot1 'slot2)))         ;instance variable names
```

The **make** method for types takes two extra arguments: a list of the types from which the new type will inherit methods, and a list of names for the new type's instance variables. The type **mycons-cell** inherits from **pair**, which has methods for high-level list-processing operations such as **map** and **print**. These methods are written in terms of the accessor operations **car** and **cdr**, so we can cause instances of **mycons-cell** to be fully functional list components simply by supplying **car** and **cdr** methods for the type:

```
(add-method (car (mycons-cell slot1) self) slot1)

(add-method (cdr (mycons-cell slot2) self) slot2)
```

Now all we have to do is define the method for initializing instances of **mycons-cell**, and then we can instantiate the type using **make**:

```
(add-method (initialize (mycons-cell slot1 slot2) self x y)
  (set! slot1 x)
  (set! slot2 y))

(make mycons-cell 1 '(2 3))
```

The **make** operation really only needs to receive one argument (the object which denotes the type that is being instantiated) but our call contains the extra values 1 and '(2 3). Before returning our new object, the code for **make** sends the object an

initialize message which includes these two values. Our **initialize** method then uses these values to set up the object's internal state.

Because **mycons-cell** has **car** and **cdr** methods and inherits all of the system's list-processing methods from the supertype **pair**, our new object acts exactly like any other cons cell in the system. For example, when we apply the **print** operation to the object, it prints out as "(1 2 3)".

Although the type **mycons-cell** merely duplicates the functionality of the existing **cons-pair** type, we could just as easily define a radically different type of cons cell, such as a variety that contains thunks which aren't evaluated until their values are actually needed. Such a type would make it possible to build potentially infinite, lazily evaluated lists. In fact, in our implementation of Oaklisp, the variable **prime-list** is bound to a list of all prime numbers that was defined using this type of lazy cons cell.

Because Oaklisp types are represented by ordinary run-time objects, it is even possible to define new kinds of types at user level. For example, we have defined the **coercible-type** type, each instance of which is a type that responds to the **coercer** operation by returning the coercion operation for that type. For example, when the expression **((coercer string) 'foo)** is evaluated, the **coercer** operation is applied to the coercible type **string**, which returns the (anonymous) operation for coercing things to strings. This operation is then applied to the symbol 'foo, and because a method for the string-coercer operation has been defined for symbols, the result of this call is the string "**foo**".

This ability to add significant new features to Oaklisp at the user level is a result of the consistency of its semantics. Everything in the Oaklisp world is a full-fledged object with a type that fits into the inheritance hierarchy, and all computation occurs as the result of generic operation applications.

Due to the language's temporal consistency, **add-method** can be done at any time, and it is thus necessary for the method dispatch mechanisms to be dynamically modifiable. This is in contrast to statically compiled implementations of languages like C++, but is similar to interactive programming environments like Common Lisp's CLOS. Since these dynamic modification mechanisms must be present for processing incremental method addition, it seems capricious to deny the programmer run-time access to them.

8.2 Memory Format

Most objects, whether system- or user-defined, are stored in a standardized "boxed" format. The first word in this format is a reference to the object's type. The type of an object not only carries semantic weight, but also permits the run-time system to determine the answer to practical questions such as the number of storage cells that are

included in a given object. The type is followed by storage for the object's instance variables.

The two objects shown here are instances of the pre-defined **cons-pair** type and a hypothetical, user-defined **lazy-cons-pair** type. Each of the objects contains a reference to the type of the object, followed by an appropriately sized block of storage for its instance variables.

Our implementation of Oaklisp currently runs only on machines with 32-bit words (although porting to a machine with slightly different word size, such as 36 or 24, should be very simple). These words are divided into two contiguous chunks: free words and allocated words. The *free pointer* points to the division between these two chunks, and it is incremented as memory is allocated. When allocating an object would push the free pointer beyond the limits of memory, a garbage collection is performed.

The allocated portion of memory is divided into *boxed objects* and *solitary cells*. Each aggregate object is a contiguous chunk of cells. The first cell of an object is a reference to its type; if the type is *variable length*, the second stores the length of the object, including the first two cells. The remainder of the cells hold the instance variables. Solitary cells are cells that are not part of any object, but are the targets of locatives, and are used heavily in the implementation of variables, discussed later.

8.2.1 Tags

The low two bits of each word contain a tag value which tells how the word should be interpreted. The most important question that the tag answers is whether a word should be interpreted as an "immediate" object or as a reference to a "boxed" object that lives elsewhere.

For example, if the tag has the value 00_2, the high 30 bits of the word are interpreted as an integer, but if the tag has the value 11_2, then the high 30 bits of the word are interpreted as a reference to the boxed object that resides at the address specified by those 30 bits.

While reserving 2 bits for a tag clearly costs us something in the case of an integer, the tag is free in the case of a reference to a boxed object because the 2 low bits of a word-aligned pointer always have the same value and hence carry no information.

Fixnums, locatives, and characters are stored as single-word, immediate objects whose types are specified by their tag bits.

31 30 29 28 27 26 ... 11 10 9 8	*7 6 5 4 3 2*	*1 0*	*type*
two's complement integer		**0 0**	fixnum
data	**subtype**	**1 0**	other immediate type
address		**0 1**	locative (pointer to cell)
address		**1 1**	reference to boxed object

To simplify garbage collection, we do not permit any exceptions to the tagging protocol. As a result, strings and code vectors are not as dense as they would be if raw binary data could be stored in memory.

8.2.2 *Efficient Access to Predefined Slots in Objects*

To justify this tagging scheme, consider what happens when the Oaklisp processor executes a **cdr** instruction, ignoring type checking for the moment. The processor first strips the tag value of 11_2 from the reference which is the instruction's argument to obtain a pointer to a memory location. The processor then fetches the contents of the second slot of the memory structure that begins at that location. An interesting optimization is possible: subtraction rather than bit-wise **and** can be used to strip the tag from the reference. Then the C expression to perform **cdr** on a reference r (an unsigned long) is

```
*((ref *)(r - 0x3) + 2)
```

which gets constant folded to

```
*(ref *)(r + 5)          /*     5 = 2*4 - 3     */
```

Most computers can use addressing modes to dereference a pointer with an offset, so the **cdr** operation can be performed by a single instruction of the form

```
ldl    5@r1 ,r2.
```

But even on computers that can't do this in one instruction, it is just as easy to add an offset of 5 as it would be to add an offset of 8 to access the cdr of a cons cell, so no efficiency is lost by using a tag value of 3 rather than a tag value of 0 for references.

The only time when a tag value of 0 would be better is when the machine must access the type field of an object. However, some machines ignore the low two bits when performing a long word fetch anyway, and on the remaining machines the advantage of avoiding tag manipulations on arithmetic seems to outweigh the overhead of slightly slower access to the type fields of boxed objects.

8.2.3 Storage Reclamation

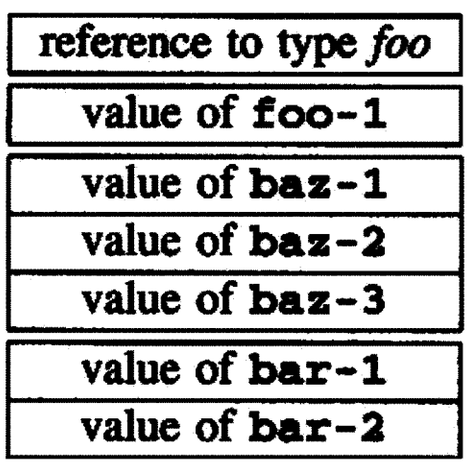
Our garbage collector is of the popular stop-and-copy variety [1]; the spaces to be reclaimed are renamed *old*, all accessible objects in the old spaces are transported to a new space, and the old spaces are reclaimed. The data present in the initial world is considered static and is not part of old space in normal garbage collections, only in *full* garbage collections, which also move everything not reclaimed into static space. Before dumping the world to a save file, a modified full garbage collection is performed in which the stacks are left out of the root set.

Locatives can point to a single cell in the middle of a large object, and the garbage collector is able to deallocate all of an object except for those cells pointed to by locatives, which become solitary cells. Due to this complication, the collector makes an extra pass over the heap. A paper with more complete details on this technique is in press.

The weak pointer table is scanned at the end of garbage collection, and references to deallocated objects are discarded. Desired new space size is changed dynamically, being expanded after a garbage collection when the allocation system judges that it would have been better to have allocated more space. The entire garbage collector is written in C, and even in code that places heavy demands on the storage allocation subsystem, such as bignum arithmetic, the time spent in the garbage collector is a tiny fraction of the total time.

The user interface to the garbage collector is quite simple. Normally, the user need not be concerned with storage reclamation, as upon the exhaustion of storage the garbage collector is automatically invoked. A switch is provided to allow the user to turn off noise messages about garbage collection, and Oaklisp operations to force normal and full garbage collections are provided.

8.2.4 Inheritance of Instance Variables



Pictured above is the memory format for an instance of a type *foo* which inherits from types *bar* and *baz*. The type *bar* has instance variables **bar-1** and **bar-2**, the type *baz* has instance variables **baz-1**, **baz-2** and **baz-3**, and *foo* has instance variable **foo-1**.

The sharp-eyed reader will note that the instance of *foo* is structured such that the instance variables associated with its constituent types *foo*, *bar*, and *baz* are stored in separate, contiguous memory blocks. Because Oaklisp methods are not allowed to contain direct references to the instance variables of supertypes, this memory format permits the compiler to implement all of the instance variable references in a method using a base-pointer-relative addressing mode. At run time, the processor's base-pointer is always set to the beginning of the instance variable block for the type which actually supplied the method which is currently being executed.

When a type inherits a type through two different routes, it only gets a single instance variable block for that type.[2] Because of our strict segregation of instance variables from the component types in a composite type, if the instance variables of two types inherited by a third have the same names, they are still distinct variables. This is in marked contrast to ZetaLisp Flavors or CLOS, in which references to instance variable must pass through mapping tables, resulting in considerable overhead. There are also important modularity considerations in favor of our scheme which are beyond the scope of this document, but are discussed in detail by Snyder [7]. Our semantics allow us to reference instance variables very quickly once the location of the relevant instance variable block for a given method has been determined. It also allows us to always use the same compiled code for a given method, regardless of whether it is being invoked upon an instance of the type for which it was originally defined or upon an instance of an inheriting type.

8.2.5 The Memory Format of Types and Operations

To show more of the character of the Oaklisp system, we will now look at the internal representation of instances of two important types. All objects, including these, have the same basic format as instances of user-defined types.

operation
`lambda?`
`cache-type`
`cache-method`
`cache-type-offset`

Operations are tokens whose only essential property is their identity. Thus, in our initial implementation, the memory format of an operation included only a single word: a reference to the **`operation`** type. Currently, operations contain some state that the

[2]This was a rather arbitrary implementation decision, and should not be relied upon by users. In fact, it will likely be changed in a future release.

run-time system uses to speed up message sending. For example, if an operation has only one method and that method is associated with the root of the type hierarchy, then it is possible to jump straight to this method whenever the operation must be performed. We indicate that an operation has this property by storing its sole method in its **lambda?** slot. (The instance variables associated with our method-caching scheme will be described later in this chapter.)

In order to allow the C code to refer to instance variables of important low-level system types as efficiently as possible, types whose instance variables are used directly by the C code are *top wired,* which forces the involved instance variables to appear first in memory. Except for the inability to inherit from more than one top-wired type at once, this top wiring is invisible to users. Most top-wired types have names like **%code-vector** or **%closed-environment**, and are not of interest to users anyway.

settable-operation
lambda?
cache-type
cache-method
cache-type-offset
the-setter

Settable operations [3] (a subtype of normal operations described below) contain two blocks of instance variables, one for the **settable-operation** type, and another for the **operation** type from which it inherits. Because **operation** is top wired, its instance variable block appears at the top of an instance of **settable-operation**.

Now we look at the memory format of a type. In particular, we will show the object which represents the **settable-operation** type. Thus, the reference in the header of the settable operation shown above points to this object:

type	
instance-length	**6**
supertype-list	**(#<Type OPERATION>)**
ivar-list	**(the-setter)**
type-bp-alist	**((#<Type SETTABLE-OPERATION> . 5)** **(#<Type OPERATION> . 1))**
operation-method-alist	**((#<Operation SETTER> . #<Method 3652>))**

As always, the first word in the object is a reference to its type, in this case, the **type** type. The second word in the object contains the value of its first instance variable, **instance-length**, which tells the system how many words of storage need to be allocated for each instance of the type. The garbage collector also uses this information when it copies instances of the type from old to new space.

The rest of the information in a type object governs the method lookup process which occurs when an operation is applied to an instance of the type. For example, the variable **`operation-method-alist`** stores the association between operations and methods for the type. However, this association list contains no information about the methods of the type's supertypes, so if a method can't be found locally for a given operation, a depth-first search of the inheritance graph is performed. The immediate supertypes of a type are recorded in the variable **`supertype-list`**.

Once a method has been found for an operation, the processor's base-pointer register is loaded with the address of the relevant instance variable block in the object that was the operation's first argument. The base pointer is set using to **`type-bp-alist`**, which describes the layout of the various instance variable blocks in instances of the object's type.

Finally, the field **`ivar-list`** specifies the layout of the variables in an instance variable block for the type. This map is used by the compiler to compute the offsets for the base-pointer-relative bytecodes that it emits for references to instance variables.

8.3 Processor Model

Our implementation of Oaklisp is based on a virtual processor that is emulated by a C program. The virtual processor contains three important registers: the program counter, the environment pointer, and the base pointer. When an operation is applied to an object by the **`funcall-cxt`** instruction, the processor pushes the values of these three registers on its context stack and then searches for a method for the operation, starting at the type of the object and continuing up the inheritance hierarchy if necessary. The **`funcall-tail`** instruction, which is identical to the **`funcall-cxt`** instruction except that it does not push anything onto the context stack, is used when the processor doesn't need to return to the current execution context and hence shouldn't save its state. The method which results from this search contains two parts: a reference to a vector of instructions, and a reference to a vector of storage cells which represents the method's lexical environment. These two references are used to set the processor's program counter and environment pointer. The base pointer is set to the top of the block of instance variables from the type the method was defined for.

8.3.1 Calling Conventions

The processor contains two stacks: one for values and one for saved processor state. This separation allows the value stack to smoothly accumulate values for function calls. Before a tail-recursive call, the parameters of the current call must be removed from the stack. Because return addresses do not have to be removed from the value stack

during this process, stack motion is reduced. We feel that this dual stack system with an unframed value stack is primarily responsible for the efficiency of function calls in our implementation.

The calling protocol is illustrated by the following code vector, which the compiler generated for the expression `(lambda (a b) (foo (bar b (baz b)) a))`. The convention for verifying that the correct number of arguments is included in each function call can also be seen here; the processor has a **`nargs`** register whose value is set by callers and checked by callees.

Bytecode	*Resulting value stack contents*	*English Description*
`(check-nargs 2)`	... b a	The lambda's args are on the stack.
`(load-stk 0 a)`	... b a a	Preload second arg for **`foo`**.
`(load-stk 2 b)`	... b a a b	Load arg for **`baz`**.
`(load-glo-con baz)`	... b a a b baz	
`(store-nargs 1)`		
`(funcall-cxt)`	... b a a Z	Call **`baz`** and return, yielding second arg for **`bar`**.
`(load-stk 3 b)`	... b a a Z b	Load first arg for **`bar`**.
`(load-glo-con bar)`	... b a a Z b bar	
`(store-nargs 2)`		
`(funcall-cxt)`	... b a a R	Call **`bar`** and return, yielding first arg for **`foo`**.
`(blt-stk 2 2)`	... a R	blow away the lambda's args
`(load-glo-con foo)`	... a R foo	
`(store-nargs 2)`		
`(funcall-tail)`		call **`foo`**, but don't return.

8.3.2 *C-Level Optimizations*

In addition to optimizations discussed elsewhere in this chapter, a number of tricks at the C level were used to speed up the bytecode emulator.

Judicious use of register declarations and manual reuse of variables sped things up considerably, as did avoiding all procedure calls in common cases, and taking care that important system variables were able to live in registers. The latter was somewhat difficult because of interactions with the garbage collector. In our solution, storage allocation was done with a macro which incremented the free pointer and checked it against the upper limit of free space. Only if no storage was available was a procedure, namely the garbage collector, called. In order to allow register variables to hold members of the root set, such variables were pushed onto a special stack before calling the garbage collector, and popped off afterwards. Similarly, the value and context stack pointers were copied from local variables to global ones before calling the garbage collector. This

allowed the context and value stack pointers, as well as emulator temporaries, to be kept in registers, which resulted in considerable speedup.

Tag checking was accomplished by macros, which were carefully tuned by benchmarking and examining the assembly level output of the compiler. Similarly, overflow checking for fixnum arithmetic was carefully tuned. In the case of overflow checking, we provided a number of different ways to detect overflows, selectable by compile time switches, because of extra constructs available on some machines. Much to our chagrin, although most machines raise an overflow flag, we were unable to find any way to access these flags, even by abandoning portable constructs. If this optimization were possible, we estimate that the tak benchmark could be sped up by about 10%.

Stack buffer overflow and underflow checking was another good target for optimization. Rather than doing bounds checking each time a value is pushed or popped, the code was modified to check only at strategic locations, where it was ensured that the stack buffer was in a state where it was appropriate to execute the entire next instruction. By making sure that there is always at least one element on the stack, unary instructions were able to dispense with buffer bounds checking entirely. Such instructions are quite common, including operators like `car` and `contents`, so this optimization alone was quite fruitful. All in all, efforts to economize on buffer bounds checking were well spent, resulting in about a 30% speed-up.

Since stack dumping or reloading procedures are called when the stack pointer exceeds the stack buffer bounds, and for speed reasons the stack pointers are kept in register variables, the stack buffer bounds checking macros load and unload the stack pointers from global structures across these procedure calls. Even more speed could be gained by using virtual memory facilities to trap at the edges of the stack buffer, obviating the need for explicit stack buffer bounds checking entirely, but at the loss of portablity.

8.4 Compilation

Our Oaklisp compiler implements the following primitive language constructs:

- variables
- combinations
- `quote`
- `if`
- `make-locative`
- `add-method`
- `labels`

All other special forms are macro-expanded in combinations of the primitive forms. For example, the familiar special forms **lambda** and **set!** are macro-expanded into combinations of function calls and **add-method** and **make-locative** forms.

8.4.1 Constant Folding and Frozen Variables

The compiler knows about the special properties of a number of operation subtypes, such as **open-coded-mixin** which informs the compiler that an operation can be open coded, **no-side-effects-mixin**, which permits constant folding, and **backward-args-mixin**, which causes the arguments to the operation to be pushed onto the stack in reverse order when the operation is open coded. These, in concert with a mechanism by which global variables can be declared *frozen,* which means that their values will never change, allow our simple compiler to perform a surprisingly wide range of optimizations.

For example, the **list** operation contains **backward-args-mixin** and **open-coded-mixin**, and is frozen. Thus, when compiling the expression **(list 1 2 3)**, the compiler first generates code to push the arguments in reverse order:

```
(load-imm 1) (load-imm 2) (load-imm 3)
```

and then calls **list**'s open-coder to obtain the following efficient bytecode sequence:

```
(load-reg nil)(reverse-cons)(reverse-cons)(reverse-cons)
```

where the **reverse-cons** instruction is a version of the **cons** instruction that takes its arguments in reverse order.

Another example is the expression **(set! (car x) y)**. This is macroexpanded to **((setter car) x y)**. Both **setter** and **car** are frozen, and the **setter** operation has **no-side-effects-mixin**, so the **(setter car)** expression is constant folded at compile time to an anonymous operation. This anonymous operation has **open-coded-mixin**, so it gets open coded as the instruction **(set-car)**, and the whole code fragment thus compiles to

```
(load-glo-con y) (load-glo-con x) (set-car).
```

8.4.2 Function Calls and Global Variables

The expansion of **lambda** makes use of the fact that all upward paths through the Oaklisp inheritance graph terminate at the distinguished type **object**, and so any method associated with this type functions as a "default" method that applies to objects of

any type. Thus, `(lambda (x y) (+ x y))` can be macro expanded to `(add-method ((make operation) (object) x y) (+ x y))`. When evaluated, this expansion generates a new anonymous operation and then associates it with a method at the top of the inheritance hierarchy.

The special form `(set! foo 3)` is macro-expanded into the combination `((setter contents) (make-locative foo) 3)`. This combination is compiled as follows. First, the compiler emits the instruction `(LOAD-IMM 3)` to load the immediate value 3 onto the stack.

Next, after determining that the variable `foo` is global, the compiler translates the special form `(make-locative foo)` into the pseudo-instruction `load-glo`, supplying the symbol `foo` as an argument. When this instruction is encountered by the Oaklisp loader, the relevant global namespace is consulted to find the memory location that is associated with the variable name `foo`. A locative to this memory cell is then created and plugged into the argument field of the instruction, which is then turned into an ordinary `LOAD-IMM` instruction.

Finally, the combination `(setter contents)` is constant folded to yield the anonymous operation for storing a value in the memory location referenced by a locative. The compiler then notices that the type of this operation includes the supertype `open-coded-mixin`, so a message is sent to the operation requesting a bytecode sequence for executing the operation directly (that is, without a function call). In this case, the operation returns the single instruction `(set-contents)`.

So, the expression `(set! foo 3)` is macro expanded to `((setter contents) (make-locative foo) 3)`, which is compiled into the bytecode sequence `(load-imm 3) (load-glo foo) (set-contents)` and then converted by the loader into `(load-imm 3) (load-imm <locative>) (set-contents)`.

Another specially marked version of `load-imm` called `load-code` is emitted when the compiler turns an `add-method` special form into a call to the `install-method` operation. The argument to this instruction is a symbolic representation of a code vector. The Oaklisp assembler turns this into a list of opcodes together with some symbolic variable-patching information, and then the Oaklisp loader resolves the variable references and constructs the actual code vector in memory. The `load-code` pseudo-instruction is then turned into an ordinary `load-imm` instruction whose argument is a reference to the code vector.

An example of the code that would be generated for an `add-method` special form is shown below. Note that the loader converts the pseudo-instruction `load-glo-con` into the instruction `load-imm-con`, which the processor treats as a `load-imm` instruction followed by the locative-dereferencing instruction `contents`.

```
(add-method (jump (frog x-pos color)
                  self distance)
  (set! x-pos (+ distance x-pos))
  (set! color 'greener))
⟹
                  ;assumed ivar map
((load-code (code (x-pos color)
                  ((check-nargs 2)
                   (load-bp 0 x-pos)
                   (load-stk 2 distance)
                   (plus)
                   (store-bp 0)
                   (pop 3)
                   (load-imm greener)
                   (store-bp 1)
                   (return))))
 (load-glo-con jump)
 (load-glo-con frog)
 (load-glo-con install-method)
 (store-nargs 3)
 (funcall-tail))
```

8.4.3 Local Function Calls and Iteration

All looping is expressed vis tail recursion, as evident from the primitive forms the compiler recognizes listed above. Thus, in order for tight loops to be compiled efficiently it is necessary for the compiler to optimize heavily certain tail-recursive constructs. In particular, when the compiler encounters a **labels** form in which the labeled procedures are only called tail recursively and never referenced in any other way, it emits the code for the labeled procedures inline and compiles calls to them as simple branches, as in the following example.

This form describes how to append lists. Since **append** is so frequently used, it was written as a local tail-recursive loop.

```
(add-method (append (pair) oldcopy b)
  (let ((newcopy (cons (car oldcopy) b)))
    (let next ((oldpair (cdr oldcopy))
               (last-newpair newcopy))
      (if (not (null? oldpair))
          (next (cdr oldpair)
                (set! (cdr last-newpair)
                      (cons (car oldpair) b)))
          newcopy))))
```

This inner loop, which seems to involve procedure calls, nevertheless compiles into efficient code, as seen in the compiler's output.

```
        ((check-nargs 2)
         (load-stk 1 b)
         (load-stk 1 oldcopy)
         (car)
         (cons)
         (load-stk 0 newcopy)
         (load-stk 2 oldcopy)
         (cdr)
label0   (load-stk 0 oldpair)
         (branch-nil else1)
         (load-stk 4 b)
         (load-stk 1 oldpair)
         (car)
         (cons)
         (load-stk 2 last-newpair)
         (set-cdr)
         (blast 2)
         (cdr)
         (branch label0)
else1    (pop 2)
         (blt-stk 1 2)
         (return)))
```

8.5 List Optimizations

Because lists are so ubiquitous in lisp, it is desirable to expend some effort speeding up access to them. On the other hand, we designed Oaklisp so that cons cells would fit naturally into an extensible type hierarchy, as the example given in Section 8.1 shows. In initial versions of the system **car** and **cdr** were not specially handled, but were regular operations like any others. Because so much time was spent in these operations, we decided to make them into instructions. These instructions are very simple: they check if the object they are being applied to is a simple cons cell, of type **cons-pair**, which is held in a special C variable to make this check fast. If its argument is of the correct type, the instruction simply returns the appropriate value. Otherwise, the instruction traps and the normal method lookup takes place.

This optimization sped the system up by at least a factor of two. However, because it was made before method caching was inserted, it is not clear how much the system would slow down if it were to be disabled at this point.

Since the standard list processing operations, like **`append!`**, **`map`**, and **`copy`**, are called so frequently, the methods for these operations were carefully hand tuned.

8.6 Bignums

Bignums were implemented very late, and we didn't put much effort into them. They are represented in signed magnitude format, with the magnitude represented as a list of base 10,000 digits, for efficient printing and ease of carry manipulation during multiplication. Rather than the usual $O(nm)$ time multiplication algorithm, where n and m are the number of digits in the two numbers being multiplied, we use an $O(nm^{0.59})$ time algorithm, where $n > m$. In addition, because bignum division is so expensive, a division cache of the last two bignums divided is kept.

8.7 Locatives

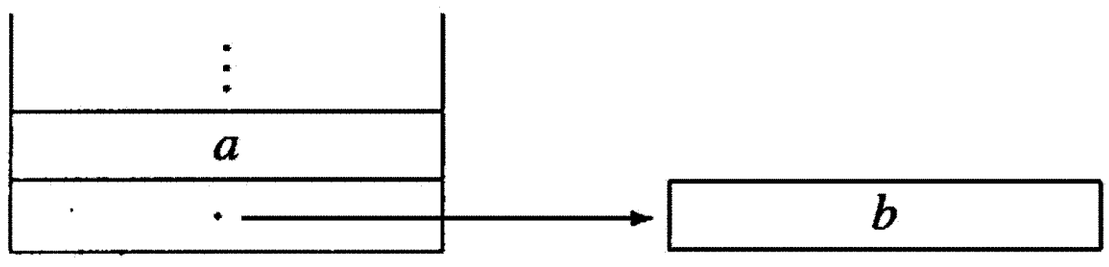
Locatives are language-level pointer objects that permit the contents of memory cells to be retrieved and modified. Locatives are created with the special form **`make-locative`** and are derefenced with the operations **`contents`** and **`(setter contents)`**.

Locatives are employed throughout the Oaklisp system. For example, locales and closures contain locatives that point to the storage cells that actually contain the values associated with variable names.

Locatives are used to keep the value stack "clean," as required by the implementation of **`call/cc`** [3]. The code that is emitted to close over or side-effect a variable causes a locative to be made to that variable. When the compiler notices that a locative is being made to a stack variable, it generates a method preamble that allocates a cell for that variable in the heap, leaving a locative to this cell on the stack instead. For example, the code fragment

```
(let ((a (f1))
      (b (f2)))
  (foo a b)
  (set! b (f3))
  (bar a b))
```

keeps **`a`** on the stack but puts the cell for **`b`** in the heap:



Locatives can be made to refer to any sort of variable, which after all are not distinguished at the language level. If one is made to an instance variable, the surrounding object can get garbage collected, leaving a solitary storage cell.

Locatives obey the following identities.

$$\texttt{(contents (make-locative } var\texttt{))} \equiv var$$
$$\texttt{(make-locative (contents } loc\texttt{))} \equiv loc$$

Although locatives complicate the garbage collector slightly, we found them so useful that we consider them well worth the price of a dedicated tag code and a somewhat slower garbage collector.

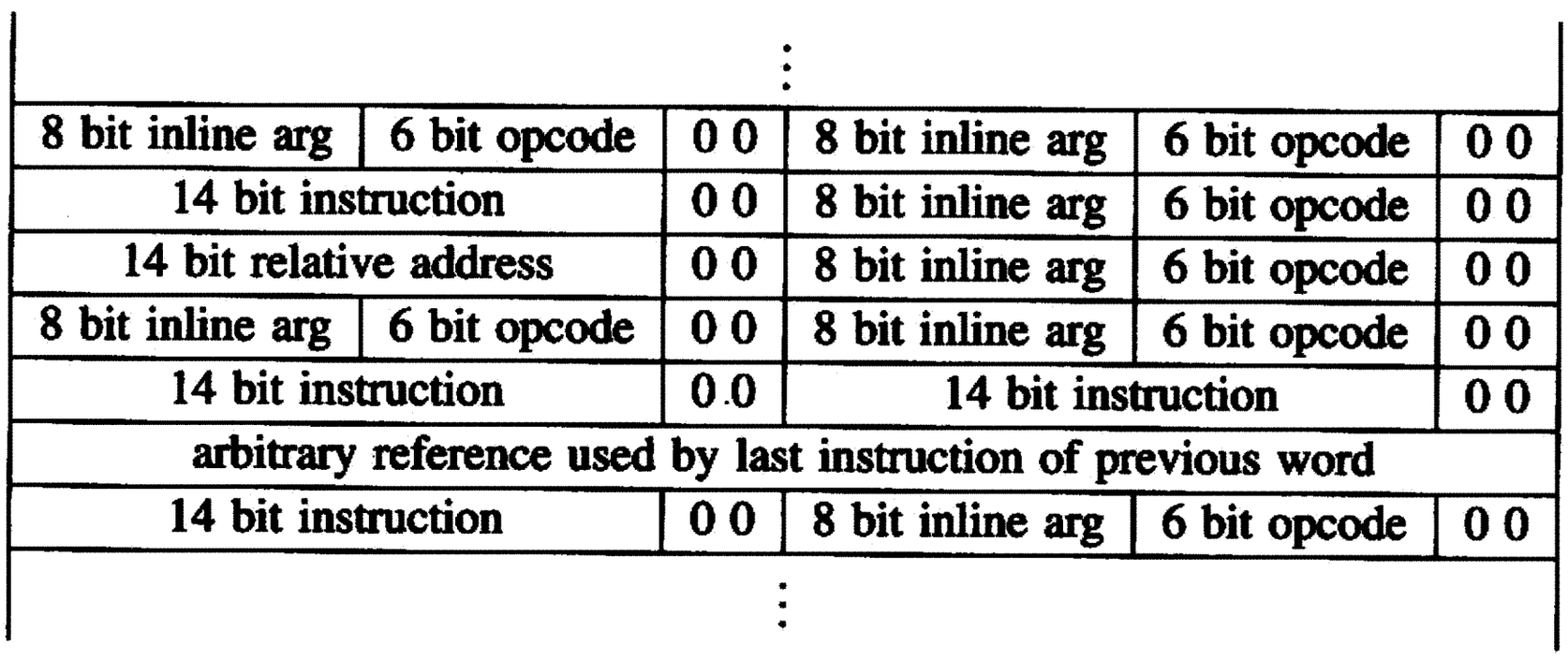
8.8 Methods

In describing how methods are created, represented, and looked up, we find ourselves concerned with references to instance variables, so we shall also describe how that works.

A method has two instance variables, one of which holds the code object that implements the method. The other contains the environment vector that holds references to lexical, or non-global, variables that were closed over. Global variable references are implemented as inline locatives to value cells.

8.8.1 Code Vectors

Code is represented by vectors of integers, which are interpreted as instructions by the bytecode emulator. This format allows code to be stored in the same space as all other objects, and allows the garbage collector to be ignorant of its existence, treating code vectors like any other vector. Bytecodes are 16 bits long, with the low 2 bits always 0. Here is an example taken from the middle of a code vector:



Note the reference to an arbitrary object in the middle of the code. To allow the garbage collector to properly handle code vectors, as well as to allow the processor to fetch the cell efficiently, this reference must be cell aligned. When the processor encounters an instruction that requires such an inline argument, if the program counter is not currently pointing to an aligned location then it is suitably incremented. This means that the assembler must sometimes emit a padding instuction, which will be ignored, between instructions that require inline arguments and their arguments.

8.8.2 Environment Vectors

An environment vector is a block of cells, each of which contains a locative to a cell. When the running code needs to reference a closed-over variable, it finds the location of the cell by indexing into the environment vector. This index is calculated at compile time, and such references consume only one instruction.

Just as it is possible for a number of methods to share the same code, differing only in the associated environment, it is also possible for a number of methods to share the same environment, differing only in the associated code, a possibility proposed by Sussman and Steele [8] and partially implemented by the Orbit compiler's *closure-hoisting* [3]. Currently the Oaklisp compiler does not generate such sophisticated constructs.

8.8.3 Invoking Methods

Methods are looked up by by doing a depth-first search of the inheritance tree. Some Oaklisp code to find a method would look like this,

```
(define (%find-method op typ)
  (let ((here (assq op (type-operation-method-alist typ))))
    (if (null? here)
        (any? (lambda (typ) (%find-method op typ))
              (type-supertype-list typ))
        (list typ (cdr here)))))
```

Once this information is found, we need to find the offset of the appropriate block of instance variables, put a pointer to the instance-variable frame in the **bp** register, set the other registers correctly, and branch.

```
(define (%send-operation op obj)
  (let ((typ (get-type obj)))
    (destructure (found-typ method) (%find-method op typ)
      (set! ((%register 'current-method)) method)
      (set! ((%register 'bp))
            (%increment-locative
```

```
          (%crunch (%data obj) %loc-tag)
          (cdr (assq found-typ (type-bp-offset-alist typ)))))
    (set! ((%register 'env)) (method-env method))
    (set! ((%register 'pc))
          (code-body-instr (method-code (%method))))))
```

Of course, the actual code to find a method is written in C and has a number of tricks to improve efficiency.

- Simple lambdas (operations which have only one method defined at the type **object**) are ubiquitous, so the overhead of method lookup is avoided for them by having a **lambda?** slot in each operation. This slot holds a zero if no methods are defined for the given operation. If the only method defined for the operation is for the type **object** then the **lambda?** slot holds that method, and the method is not incorporated in the **operation-method-alist** of type **object**. If neither of these conditions holds, the **lambda?** slot holds **#f**.

- To reduce the frequency of full-blown method lookup, each operation has three slots devoted to a method cache. When *op* is sent to *obj*, we check if the **cache-type** slot of *op* is equal to the type of *obj*. If so, instead of doing a method search and finding the instance-variable frame offset, we can use the cached values from **cache-method** and **cache-offset**. In addition, after each full-blown method search, the results of the search are inserted into the cache.

  The method cache can be completely disabled by defining **NO_METH_CACHE** when compiling the emulator. We note in passing that we have one method cache for each operation. In contrast, the Smalltalk-80 [2] system has an analogous cache at each call point. We know of no head-to-head comparison of the two techniques, but suspect that if we were to switch to the Smalltalk-80 technique we would achieve a higher hit rate at considerable cost in storage.

- In order to speed up full-blown method searches, a move-to-front heuristic reorders the association lists inside the types. In addition, the C code for method lookup was tuned for speed, is coded inline, and uses an internal stack to avoid recursion.

8.8.4 Adding Methods

A serious complication results from the fact that the type field in an **add-method** form is not evaluated until the method is installed at run time. Since the target type for the method is unknown at compile time, the appropriate instance-variable map is also unknown, and hence the correct instance-variable offsets cannot be determined. Our solution is to have the compiler guess the order (by attempting to evaluate the type

expression at compile time) or simply invent one, compile the offsets accordingly, and incorporate this map in the header of the emitted code block. When the **add-method** form is actually executed at run time, the assumed instance-variable map is compared to the actual map for the type that is the recipient of the method, and the code is copied and patched if necessary. The code only needs to be copied in the rare case when a single **add-method** is performed on multiple types that require different offsets.

After instance-variable references in the code block have been resolved (which usually involves no work at all since the compiler almost always guesses correctly) the method can actually be created and installed. Creating the method involves pairing the code block with an appropriate environment vector containing references to variables that are in the lexical environment. Because this environment vector is frequently empty, a special empty environment vector is kept in the global variable **%empty-environment** so a new one doesn't have to be created on such occasions. All other environment vectors are created by pushing the elements of the environment onto the stack and executing the **make-closed-environment** opcode. With the exception of the empty environment, environment vectors are not shared.

After the method is created it must be installed. The method cache for the involved operation is invalidated, and the method is either put in the **lambda?** slot of the operation or the **operation-method-alist** of the type it is being installed in. If there is already a value in the **lambda?** slot and the new method is not being installed for type **object**, the **lambda?** slot is cleared and the method that used to reside there is added to the **operation-method-alist** of type **object**.

(%install-method-with-env *type operation code-body environment***)** *Operation*
This flushes the method cache of *operation*, ensures that the instance-variable maps of *code-body* and *type* agree (possibly by copying *code-body* and remapping the instance variable references), creates a method out of *code-body* and *environment*, and adds this method to the **operation-method-alist** of *type*, modulo the simple lambda optimization if *type* is **object**.

Some simplified variants of this are provided, both to optimize these specialized calls because of the extra assumptions about the arguments, and to save code volume in the callers.

(%install-method *type operation code-body***)**
≡ **(%install-method-with-env** *type operation code-body*
%empty-environment)

(%install-lambda-with-env *code-body environment***)**
≡ **(%install-method-with-env object (make operation)** *code-body environment***)**

(%install-lambda *code-body***)**
≡ **(%install-method-with-env object (make operation)** *code-body*
%empty-environment)

8.9 Stacks and Continuations

8.9.1 Stack Implementation

Although the value and context stacks are logically contiguous, they are sometimes physically noncontiguous. The instructions all assume that stacks are stored in a designated chunk of memory called the stack buffer. They check if they are about to overflow or underflow the stack buffer, and if so they take appropriate actions to fill or flush it, as appropriate, before proceeding.

If the stack buffer is about to overflow, most of it is copied to a *stack segment* that is allocated on the heap. These segments form a linked list, so upon stack underflow the top segment is removed from this list and copied back to the stack buffer.

There is one more circumstance in which the stack buffer is flushed. The `call/cc` construct of Scheme [6] is implemented in terms of *stack photos,* which are snapshots of the current state of the two stacks. A `FILL-CONTINUATION` instruction forces the stack buffers to be flushed and copies references to the linked lists of overflow segments into a continuation object.

Actually, in the above treatment we have simplified what happens when a stack buffer is flushed. The emulator constant `MAX_SEGMENT_SIZE` determines the maximum size of any flushed stack segment. When flushing the stack, if the buffer has more than that number of references then it is flushed into a number of segments. This provides some hysteresis, speeding up `call/cc` by taking advantage of coherence in its usage patterns. A possibility opened by our stack buffer scheme, which we do not currently exploit, is to use virtual memory faults to detect stack-buffer overflows, thus eliminating the overhead of explicitly checking for stack overflow and underflow.

As a historical note, an early version of Oaklisp did not use a stack buffer but instead implemented stacks as linked lists of segments which were always loacted in the heap. When exceeding the top of a segment, a couple of references were copied from the top of that segment onto a newly allocated segment, providing sufficient hysteresis to prevent inordinate overhead from repeated pushing and popping along a segment boundary. Regrettably, substantial storage is wasted by the hysteresis and the overflow and underflow limits vary dynamically whereas in the new system these limits are C link-time constants. Presumably due to these factors, timing experiments between the old system and the new system were definitively in favor of the new system.

8.9.2 Catch and Throw

We provide two different escape facilities: `call/cc` and `catch`. The `call/cc` construct is that described in the Scheme standard [6], and its implementation is described

above. The `catch` facility provides the user with a second class *catch tag*, which is valid only within the dynamic extent of the `catch`.

The implementation of catch tags is very simple: they contain heights for the value and context stacks. When a catch tag is thrown to, the value and context stacks are truncated to the appropriate heights. The slot `saved-wind-count` is used for unwind protection and `saved-fluid-binding-list` is used for fluid variables.

type: escape-object
value stack height: 25
context stack height: 19
saved wind count: 3
saved fluid binding list: `((print-length . #f) ...)`

Actually, there are two variants of `catch`. In the regular variant, which is compatible with T [3], the escape object is invoked by calling it like a procedure, as in `(catch a (+ (a 'done) 12))`. In the other variant, the escape object is not called but rather thrown to by using the `throw` operation, as in `(native-catch a (+ (throw a 'done) 12))`. Although the latter construct is slightly faster, the real motivation for its inclusion is to remind the user that the the escape object being thrown to is not first class. In order to ensure that an escape object is not used outside of the extent of its dynamic validity, references to them should not be retained beyond the appropriate dynamic context.

8.9.3 Fluid Variables and Unwind Protection

Fluid (or *dynamic* or *special,* depending on your background) variables are provided using the special forms `(fluid x)` to reference them and `bind` to bind them. Fluid variables are implemented with an association list, and constructs which break the normal flow of execution, such as `throw` or `call/cc`, restore the appropriate fluid-binding list when control is transfered nonlocally.

Another facility, unwind protection, is also provided. This allows chunks of code to be executed upon entry or exit from a particular dynamic context (`call/cc` allows a dynamic context to be reentered even after it has been exited.) The fluid-binding list could be maintained with the unwind protection facility, but for efficiency reasons we implemented it separately. The unwind protection actions form a tree, and each time a nonlocal transfer of control is made, either by `throw` or by invoking a continuation, the unwind protection entries along the path from the source to the destination are executed. Care is taken to restore the appropriate fluid-variable binding list for each unwind-protection action.

All of this is done at nearly user level, so the underlying primitive mechanisms for nonlocal transfer of control need not be concerned with either fluid variables or with unwind protection. In a multprocessor version of the implementation, a fluid binding list would have to be stored for each process. We considered adding a cache for fluid variables to avoid the overhead of looking them up, but such a tiny fraction of the system's time is spent looking up fluid variables on the fluid binding list that we decided it was not worth it.

8.10 Traps

Some operations, like `car` and `contents`, are open coded as calls to bytecodes. If the operand passed to the operation is not of the precise type expected by the system, it is necessary for a full-blown operation dispatch to be performed. In cases like this, a table containing a trap operation for each instruction is indexed, and the appropriate operation is called, after setting things up so that the instruction following the trapping instruction will be the next one executed when the trap code returns. This allows users to define freely methods for system operations for their own types, even when the system operation is open coded.

One issue that arises in this context is tail recursion. If `car` were not open coded, when it occured in a tail recursive position it would be coded as `(... (load-imm car) (funcall-tail))`. With `car` open coded, the code is `(... (car)(return))`. But if the `car` instruction traps, pushing the context of the trap point onto the context stack would make the call non-tail-recursive. To avoid this, when an instruction traps if the next instruction is a return then no context is pushed onto the context stack. Similar special cases are needed when a `funcall-tail` instruction traps.

In the above treatment we have discussed synchronous traps. Another class of traps are user interrupts, used to terminate infinite loops and the like. User interrupts set a flag, which is polled by certain instructions, such as branches, carefully chosen to interrupt any loop. By using polling, we avoid the overhead of determining where in the C code the program was when the signal was fielded, and then having to clean up memory to restore the heap and stack invariants. The polling solution has also proven quite portable.

8.11 Anonymity and Printed Representations

Oaklisp objects are anonymous, but when an object is stored in a global variable, it can be accessed through that variable.

When printing an object, the default print method tries to provide an expression that will evaluate to that object, such as `car` or `(setter contents)`. These expressions

are generated relative to the current locale, and they are cached and rechecked for validity every time they are used.

When this fails, a *weak* (or non-garbage-collector-proof) pointer to the object is generated. These weak pointers are represented by small integers. To determine the value of `#<Object 427>`, evaluate the expression `(object-unhash 427)`. Weak pointers are also used in hash tables, so that an object which appears only as a key in a hash table can be deallocated by the garbage collector. For convenience, the `describe` operation, which describes arbitrary objects, and dereferences any weak pointers it is passed.)

This strategy of moving responsibility for finding names for objects into the printer avoids a trick used in most dialects of Scheme, in which many objects are created with a "name" slot which contains a symbol corresponding to the global variable in which the object in question is stored. This name slot is typically used by the printer, thus giving descriptive names to objects; but such a strategy requires storage for keeping redundant information, is susceptable to inconsistency as the system evolves, and subverts the anonymous spirit of Scheme.

8.12 Bootstrap and Portability Issues

8.12.1 Building the world

The Oaklisp world is built from files that define all of the types, operations, and other data structures that a user expects to have predefined. The most primitive of these, those which are necessary in order to load compiled files, are linked by an offline program into a cold world file that contains one huge method which builds the world from ground zero when invoked.

The cold-world linker also has to lay out a few skeletal data structures for quoted lists and symbols that appear as program constants, along with information about these structures are located so that they can be back-patched (with correct type descriptors, for example) after the world comes up. Also, a locale is built to provide access to the global environment that was implicitly in effect while the world was booting up.

The files that define the root of the type hierarchy, such as `type`, `object`, and `operation`, are carefully written using only operations that are compiled straight into bytecodes, because no function calls or `add-method`'s can occur until the machinery has been built to support them.

8.12.2 Endianity

The logical order of the instructions in a code vector depends on the byte order of the CPU running the emulator. If the machine is big-endian, i.e. addresses start at the most

significant end of a word and go down (e.g. 68000 or IBM 370 series) then instructions are executed left to right. Conversely, on a little-endian machine (e.g. a VAX) instructions are executed right to left. Of course, the Oaklisp loader has to be able to pack instructions into words in the appropriate order. The format of cold world loads is insensitive to endianity, but binary world loads are sensitive to it, so binary worlds are distributed in both big endian (with extensions beginning with `.ol`) and little endian (with extensions beginning with `.lo`) versions. When a running Oaklisp loads a compiled file, a special instruction, `%big-endian?`, tells the running Oaklisp how to pack the instructions.

8.12.3 Strings

Characters are packed into strings more densely than one character per reference, so strings are not just vectors with odd print methods; they also have accessor methods which unpack characters from their internals. Unfortunately, it is not possible to pack four eight bit characters into a single reference without violating the memory format conventions by putting something other than `0 0` in the tag field. We could pack four seven bit characters into each reference, but some computers use eight bit fonts, and the characters within the string would not be aligned compatibly with C strings anyway. We therefore use a somewhat wasteful format, which is little endian regardless of the endianity of the host. Here we document it by example, showing how the string `"Oaklisp Rules!"` is represented:

31 ... 26	25 ... 18	17 ... 10	9 ... 2	1 0
string				
object length: 8				0 0
string length: 14				0 0
0 0 0 0 0 0	`#\k`	`#\a`	`#\O`	0 0
0 0 0 0 0 0	`#\s`	`#\i`	`#\l`	0 0
0 0 0 0 0 0	`#\R`	`#\space`	`#\p`	0 0
0 0 0 0 0 0	`#\e`	`#\l`	`#\u`	0 0
0 0 0 0 0 0	`#\null`	`#\!`	`#\s`	0 0

The unused high bits of each word are set to zero to simplify equality testing and hash key computation. No trailing null character is required, although one is present two thirds of the time due to padding. When interfacing to C routines that require string arguments, such as when opening files, a special translation routine written in C is used to convert Oaklisp strings to C strings.

The representation of strings is probably the first thing that would be changed if facilities were added to permit raw binary data to exist in memory. Since the bulk of

I/O time is spent doing string manipulation, they could be a fruitful source of useful optimizations, especially considering that they have not been optimized at all yet. The easiest thing to do would be to move manipulation of simple strings into C in the same way that manipulation of simple cons cells was moved into C, but if strings were being optimized it would probably be worth modifying the garbage collector to handle raw binary data first so that they could be stored in a C-compatible fashion. Time constraints prevented us from experimenting with such measures.

8.13 Getting a Copy

Copies of the Oaklisp language and implementation manuals can be obtained by sending a request to

Catherine Copetas
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

or **Catherine.Copetas@CS.CMU.EDU** by computer mail. The most recent released version of Oaklisp is available for FTP from the host **DOGHEN.BOLTZ.CS.CMU.EDU** (**128.2.222.37**), user **anonymous**, with no particular password. The proper file to retrieve is **oaklisp/release.tar.Z**, which is a compressed tar file. This file is binary so you must put FTP into binary mode before transferring it. For those without access to FTP, a tape can be obtained by making suitable arrangements with Catherine Copetas. There is a distribution fee.

8.14 Conclusions

The Oaklisp implementation effort was a success. In less than one year of full-time work, a team of two experienced programmers implemented not just new language features, with new techniques for their efficient implementation, but also the remainder of a full featured Scheme, from rationals to bignums to hash tables. Through judicious design decisions, with feedback from repeated profiling of the evolving implementation, speed rivaling (and sometimes even surpassing!) that of contemporary native-code implementations was obtained in a portable implementation.

As a result of our experience, we have come to the strong conclusion that most Lisp implementation efforts spend a great deal of time optimizing portions of the implementation that are rarely used, and spend insufficient time worrying about the tradeoff between access to processor resources within procedures and speed of procedure calls. Our watchwords were *profile* and *experiment.* We implemented dozens of "optimizations" which

we were sure would speed the system up, only to remove them after profiling revealed the inadequacy of our intuition.

Another trick we used constantly was that of amortized optimization: techniques which save time on the average. Caching is such an optimization, as it slows down the worst case in order to speed up the average case. Our stack fragmentation technique is another example. In the worst case, `call/cc` needs to copy the entire stack onto the heap, which can take unbounded time. But if `call/cc` is used frequently, most of the stack has already been copied to the heap, so making the new continuation is cheap. The bring-to-front heuristic for method lookup is yet another. Almost every time we added an optimization in this class, we observed a speedup.

Because of our choice of memory formats, it would be a straightforward task to add a native code compiler if extreme speed was desired on a particular platform. If this implementation is to fill more than its current niche as a reasonably fast Scheme system that can be ported quickly to a new machine, to be used until native code implementation are retargetted, it will be necessary to add such native-code back ends.

References

[1] R. R. Fenichel and J. C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11), November 1969.

[2] A. J. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[3] D. Kranz. *Orbit: An optimizing compiler for Scheme*. PhD thesis, Yale University, 1988.

[4] K. J. Lang and B. A. Pearlmutter. Oaklisp: an object-oriented Scheme with first class types. In *ACM Conference on Object-Oriented Systems, Programming, Languages and Applications*, pages 30–37, September 1986.

[5] K. J. Lang and B. A. Pearlmutter. Oaklisp: an object-oriented dialect of Scheme. *Lisp and Symbolic Computation*, 1(1):39–51, May 1988.

[6] J. A. Rees, W. Clinger, et al. The revised^3 report on the algorithmic language scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986.

[7] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *ACM Conference on Object-Oriented Systems, Programming, Languages and Applications*, pages 38–45, September 1986.

[8] G. L. Steele Jr. Lambda: the ultimate declarative. Technical Report AI Memo 379, MIT AI Lab, 1976.