The
Oaklisp Implementation Guide

October 21, 2002

DRAFT

Barak A. Pearlmutter
Dept of Computer Science, FEC 313
University of New Mexico
Albuquerque, NM 87131
bap@cs.unm.edu

Kevin J. Lang
NEC Research Institute
4 Independence Way
Princeton, NJ 08540
kevin@research.nj.nec.com

Contents

Chapter 1

Introduction

This document describes the internals of the CMU implementation of Oaklisp. Although this implementation is designed for portability through the use of a bytecode interpreter written in C, the fundemental data structures and memory formats would also be suitable for a high performance implementation. In spite of the fact that Oaklisp has the potential performance penalty of being uniformly object-oriented, this implementation has proven more than competitive with other bytecode based implementations of Scheme, such at MIT's CScheme and Semantic Microsystems' MacScheme. An abbreviated version of some of the information presented here is available as a book chapter [2].

1.1 Disclaimer

Warning: this document may contain inaccuracies, and it lags behind the implementation as the system evolves.

Chapter 2

Language

This document is a description of one particular implementation of Oaklisp, and therefore contains information that is subject to change and may not be significant to users in any case. For a description of the language that does not contain a lot of arbitrary distinctions, refer to *The Oaklisp Language Manual*.

2.1 Special Forms

These special forms all work by magic, and can't be redefined or shadowed.

`(%quote` *x*`)` *Special Form*

Does what you would expect.

`(%if` *predicate consequent alternative*`)` *Special Form*

Does what you would expect.

`(%labels ((`*variable value*`)...)`*form*`)` *Special Form*

If all references to the labels are calls from tail recursive positions, this is compiled using jumps. Otherwise, it is rewritten using `let` and `set!`.

`(native-catch` *variable* `.` *body*`)` *Special Form*

Evaluates *body* within the lexical scope of *variable*, which is bound to a catch tag that is valid within the dynamic scope of this form. This is actually macro expanded to something pretty wierd.

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

Yields the specified method object.

`(%make-locative` *variable*`)` *Special Form*

Returns a locative pointing to *variable*.

`(%block .` *forms*`)` *Special Form*

Making this a primitive special form simplifies the compiler.

2.2 Macros

Most constructs that users think of as primitive are actually macros. This simplifies the compiler by both reducing the number of special forms to be handled and eliminating the need for the compiler to check whether special forms it encounters are syntactically correct.

`(quote` *x*`)` *Macro*
≡ `(%quote` *x*`)`

`(add-method (`*operation* `(`*type* `.` *ivar-list*`) .` *arg-list*`) .` *body*`)` *Macro*
This turns into `%add-method`, filling in the default type and putting a block around the body if necessary.

`(lambda` *arglist* `.` *body*`)` *Macro*
≡ `(add-method ((make operation) .` *arglist*`) .` *body*`)`
Functions are made by hanging methods off of `object`. There is an optimization in the compiler that expands car-position lambdas inline.

`(catch` *var* `.` *body*`)` *Macro*
≡ `(native-catch` *x* `(let ((`*var* `(lambda (`*y*`) (throw` *x* *y*`)))) .` *body*`))`

`(define` *symbol value*`)` *Macro*
≡ `(set!` *symbol value*`)`

`(define (fluid` *symbol*`)` *value*`)` *Macro*
≡ `(set! (fluid` *symbol*`)` *value*`)`

`(define (`*variable* `.` *arglist*`) .` *body*`)` *Macro*
≡ `(set!` *variable* `(lambda` *arglist* `.` *body*`))`

`(set!` *symbol value*`)` *Macro*
≡ `(set! (contents (make-locative` *symbol*`))` *value*`)`

`(set! (`*op* $a_1 \ldots a_n$`)` *value*`)` *Macro*
≡ `((setter` *op*`)` $a_1 \ldots a_n$ *value*`)`

`(set` *location value*`)` *Macro*
An obsolete form with semantics identical to `set!`.

`(make-locative` *symbol*`)` *Macro*
≡ `(%make-locative` *symbol*`)`

`(make-locative (`*op* `.` *args*`))` *Macro*

≡ `((locater` *op*`)   .` *args*`)`

`(if` *test thenform*`)` *Macro*
≡ `(%if` *test thenform* `(undefined-value))`

`(if` *test thenform elseform*`)` *Macro*
≡ `(%if` *test thenform elseform*`)`

`(fluid` *symbol*`)` *Macro*
≡ `(%fluid (quote` *symbol*`))`

`(bind-error-handler )` *Macro*
See the language manual for a semantic definition.

`(catch-errors )` *Macro*
Implemented with `bind-error-handler` and `native-catch`.

`(bind (((fluid` *symbol*`)` *value*`)...) .` *body*`)` *Macro*
Implemented using `let` and `set!`. Hacks to `native-catch` and `call/cc` are also necessary. Essentially, the bindings are pushed onto `fluid-bindings-alist` for the dynamic scope of the `bind`. For details, see section 7.1.

`(wind-protect` *before form after*`)` *Macro*
≡ `(dynamic-wind (lambda ()` *before*`) (lambda ()` *form*`) (lambda () ` *after*`))`

`(funny-wind-protect` *before abnormal-before form after abnormal-after*`)` *Macro*
A `wind-protect` evaluates *before*, *form*, and *after*, returning the value of *form*. If *form* is entered or exited abnormally (due to `call/cc` or `catch`) the *before* and *after* forms, respectively, are automatically executed. `funny-wind-protect` is the same except that different guard forms are evaluated depending on whether the dynamic context is entered or exited normally or abnormally.

The following macro definitions may be found in *The Revised^3 Report on Scheme* [3].

`(let )` *Macro*

`(let* )` *Macro*

`(cond )` *Macro*

`(or )` *Macro*

`(and )` *Macro*

2.3 Primitive Types

The following types are *immediates*. They have no instance variables, occupy no heap storage, and are directly manipulated by the micro-engine. Their references have special tag bits. See section 3.1.

`fixnum` *Type*

`character` *Type*

`locative` *Type*

2.4 Open-Coded Operations

Because arithmetic on `fixnums` is so common, a special mechanism is used to perform operations for which byte-codes exist. When the compiler sees one of these operations in a program, it emits the corresponding byte-codes inline. At run-time, the micro-engine checks the tag-bits of the operands to verify that they are `fixnums`. If they are, the arithmetic is performed immediately. Otherwise, a hardware trap occurs which causes the usual search up the type hierarchy to find the appropriate method to perform the operation. The only restriction this places on the full generality of the usual method system is that new methods cannot be defined for the simple arithmetic operations on `fixnums`.

The operations which fall under this restriction are the following:

`(zero?` *number*`)` *Operation*

`(!=` *number1 number2*`)` *Operation*

`(*` *number* ...`)` *Operation*

`(+` *number* ...`)` *Operation*

`(-` *number1 number2* ...`)` *Operation*

`(1+` *number*`)` *Operation*

`(<` *number1 number2*`)` *Operation*

`(<=` *number1 number2*`)` *Operation*

`(=` *number1 number2*`)` *Operation*

`(>` *number1 number2*`)` *Operation*

`(>=` *number1 number2*`)` *Operation*

`(ash-left` *integer1 integer2*`)` *Operation*

`(ash-right` *integer1 integer2*`)` *Operation*

`(bit-and` *integer1 integer2*`)` *Operation*

`(bit-andca` *integer1 integer2*`)` *Operation*

`(bit-equiv` *integer1 integer2*`)` *Operation*

`(bit-nand` *integer1 integer2*`)` *Operation*

`(bit-nor` *integer1 integer2*`)` *Operation*

`(bit-not` *integer*`)` *Operation*

`(bit-or` *integer1 integer2*`)` *Operation*

`(bit-xor` *integer1 integer2*`)` *Operation*

`(object-unhash` *integer*`)` *Operation*

`(positive?` *number*`)` *Operation*

`(quotient` *number1 number2*`)` *Operation*

`(rot-left` *fixnum1 fixnum2*`)` *Operation*

`(rot-right` *fixnum1 fixnum2*`)` *Operation*

`(minus` *number*`)` *Operation*

`(modulo` *number1 number2*`)` *Operation*

`(negative?` *number*`)` *Operation*

The following operations are also open-coded and take type-mismatch traps if necessary. They can be add-method'ed to, but only for types that are not handled by the microcode. It should be clear from the discussion below which types the bytecode expects.

`(throw` *tag value*`)` *Operation*

Causes control to return from the `native-catch` form that generated *tag*.

`(contents` *locative*`)` *Locatable Operation*

Dereferences *locative*. `((setter contents)` *locative value*`)` puts *value* in the cell pointed to by *locative*.

`(object-unhash` *fixnum*`)` *Operation*
Returns the object that the weak pointer *fixnum* points to, or `#f` if the object has been reclaimed by the garbage collector.

The following operations are open-coded, and the microcode can handle objects of any type, so they can't be `add-method`'ed.

`(get-type` *object*`)` *Operation*
Returns the type of *object*.

`(eq?` *x y*`)` *Operation*
Determines whether *x* and *y* are the same object. Implemented by checking if the references are identical.

`(object-hash` *x*`)` *Operation*
Returns a "weak pointer" to *x*.

`(cons` *x y*`)` *Operation*
Conses *x* onto *y* in the usual lisp fashion.

`(identity` *x*`)` *Operation*
Returns *x*.

`(list .` *args*`)` *Operation*
Constructs a list; `(list` *a b c*`)` ≡ `(cons` *a* `(cons` *b* `(cons` *c* `'()))).`
Actually, the `list` operation is open coded and has `backwards-args-mixin` mixed into the type, so its arguments are pushed onto the stack in left to right order. The code emitted for the operation itself is just a `(load-reg nil)` followed by a bunch of `reverse-cons` instructions, one for each argument.

`(list*` $a_1 \ldots a_n$`)` *Operation*
≡ `(cons` $a_1 \ldots$ `(cons` a_{n-1} a_n`) ...).`
This is open coded in nearly the same way as `list`.

`(not` *x*`)` *Operation*
≡ `(eq?` *x* `#f)`

`(null?` *x*`)` *Operation*
≡ `(eq?` *x* `'())`

`(second-arg` *x y* `.` *rest*`)` *Operation*
Returns *y*. Remember, Oaklisp does not guarantee any particular order of evaluation of arguments.

The following operations are open-coded, but the microcode traps out if the arguments are not simple cons cells. They can not be `add-method`'ed to for the type `cons-pair`.

`(car` *pair*`)` *Locatable Operation*

`(cdr` *pair*`)` *Locatable Operation*

`(caar` *pair*`)` *Locatable Operation*

`(cadr` *pair*`)` *Locatable Operation*

`(cdar` *pair*`)` *Locatable Operation*

`(cddr` *pair*`)` *Locatable Operation*

`(caaar` *pair*`)` *Locatable Operation*

`(caadr` *pair*`)` *Locatable Operation*

`(cadar` *pair*`)` *Locatable Operation*

`(caddr` *pair*`)` *Locatable Operation*

`(cdaar` *pair*`)` *Locatable Operation*

`(cdadr` *pair*`)` *Locatable Operation*

`(cddar` *pair*`)` *Locatable Operation*

`(cdddr` *pair*`)` *Locatable Operation*

`(caaaar` *pair*`)` *Locatable Operation*

`(caaadr` *pair*`)` *Locatable Operation*

`(caadar` *pair*`)` *Locatable Operation*

`(caaddr` *pair*`)` *Locatable Operation*

`(cadaar` *pair*`)` *Locatable Operation*

`(cadadr` *pair*`)` *Locatable Operation*

`(caddar` *pair*`)` *Locatable Operation* `(cadddr` *pair*`)` *Locatable Operation*

`(cdaaar` *pair*`)` *Locatable Operation*

`(cdaadr` *pair*`)` *Locatable Operation*

`(cdadar` *pair*`)` *Locatable Operation*

`(cdaddr` *pair*`)` *Locatable Operation*

`(cddaar` *pair*`)` *Locatable Operation*

`(cddadr` *pair*`)` *Locatable Operation*

`(cdddar` *pair*`)` *Locatable Operation*

`(cddddr` *pair*`)` *Locatable Operation*

2.5 Subprimitives

The following operations should be used only deep within the system. Unless otherwise noted below, when a subprimitive encounters a domain error normal Oaklisp code is not trapped to. Rather, you're lucky if the system dumps core.

`(%assq` *object alist*`)` *Operation*
Does the usual association list lookup, but assumes that *alist* is made out of simple cons pairs. Passing it lazy lists or things like that will crash the system.

`(%big-endian?)` *Operation*
Returns `#t` or `#f` depending on whether instructions are ordered starting at the high half of a reference or the low half of a reference, respectively. On all machines that I know of, this is the same as the endianity of bytes.

`(%continue` *stack-photo*`)` *Operation*
Resumes *stack-photo*, abandoning the current stack.

`(%fill-continuation` *empty-stack-photo*`)` *Operation*
Fills in the template stack snapshot *empty-stack-photo* with the appropriate information, copying sections of the stack into the heap where necessary, and returns its argument.

`(%filltag` *empty-catch-tag*`)` *Operation*
Fills in *empty-catch-tag* with the current stack heights.

`(%crunch` *data tag*`)` *Operation*

Returns a reference with the data portion *data* and a tag of *tag*. Traps if either argument is not a fixnum.

`(%data` *x*`)` *Operation*

Returns the non-tag field of *x* as a fixnum.

`(%tag` *x*`)` *Operation*

Returns the tag of *x* as a fixnum.

`(%gc)` *Operation*

Forces an immediate normal garbage collection.

`(%full-gc)` *Operation*

Forces an immediate full garbage collection. At the end of the full garbage collection, new space size is set back to its original value.

`(%get-length` *x*`)` *Operation*

Returns the number of storage cells occupied by *x*. Zero for immediates.

`(%increment-locative` *locative n*`)` *Operation*

Returns a locative to the cell *n* beyond the cell pointed to by *locative*.

`(%load-bp-i` *n*`)` *Locatable Operation*

Loads the contents of self's instance variable number *n*. Not for the squeamish, as who is really "self" and who would be self except that the compiler is compiling away intermediate lambdas is very implementation specific.

`(%make-cell` *value*`)` *Operation*

Returns a locative to a new cell containing *value*. Could be defined with `(define (%make-cell x) (make-locative x))`.

`(%make-closed-environment` $a_1 \ldots a_n$`)` *Operation*

Returns a new environment containing $a_1 \ldots a_n$. At least one object is required. To get an empty environment, look in `%empty-environment`.

`(%print-digit` *n*`)` *Operation*

Prints *n* as a single decimal digit to `stdout`. Used to indicate various error conditions during the boot process.

`(%push . ` *args*`)` *Operation*

Pushes *args* onto the stack, returning (so to speak) the leftmost argument. This would be about the same as `values`, if we had multiple value return.

`(%read-char)` *Operation*

Returns a character read from `stdin`. No buffering. For use by the cold load stream.

`(%return)` *Operation*

Generates the `return` bytecode. Doesn't push anything onto the stack. Will corrupt the stack unless you really know what you are doing.

`(%allocate` *type size*`)` *Operation*

Allocates a block of storage *size* long, filling in the type field with *type*. *Type* should not be variable length.

`(%varlen-allocate` *type size*`)` *Operation*

Allocates a block of storage *size* long, filling in the type field with *type* and the size field with *size*. *Type* should be a variable length type. Using this instead of `%allocate` where appropriate avoids a window of gc vulnerability.

`(%write-char` *char*`)` *Operation*

Writes the character *char* to `stdout`. No buffering or anything.

`(%↑super-tail` *type operation object*`)` *Operation*

Generates the `↑super-tail` bytecode, passing it appropriate arguments. This is used only used in the implementation of `↑super`. Once the compiler is modified to handle the `↑super` construct directly this will no longer be needed.

2.6 Defined Types

The following types are completely defined in Lisp.

`object` *Type*

This type is the top of the inheritance hierarchy. Ordinary functions are installed as methods for this type.

`type` *Type*

New types are generated by instantiating this type.

`variable-length-mixin` *Type*

This mixin allows each instance of a type to have a vector of anonymous cells tacked on the end. It also provides several low-level methods for indexed references into such vectors. Currently, the only variable-length types are `vector`, `%code-vector` and `%closed-environment`.

`open-coded-mixin` *Type*

If this is mixed in to an operation, the compiler will send it a `get-byte-code-list` message, and use the result instead of a regular function call whenever the operation appears in a program.

`pair` *Type*

`cons-pair` *Type*

`null-type` *Type*

`vector` *Type*

`operation` *Type*

`settable-operation` *Type*

`locatable-operation` *Type*

`%method` *Type*

`%code-vector` *Type*

`%closed-environment` *Type*

`locale` *Type*

`general-error` *Type*

`foldable-mixin` *Type*

2.7 Defined Operations

The methods for these operations are written in low level Oaklisp.

(`apply` *operation* $a_1 \ldots a_n$ *arglist*) *Operation*

Calls *operation* with arguments $a_1 \ldots a_n$ and the contents of *arglist*. For instance, `(apply + 1 2 '(3 4))` $\Rightarrow$ `10`.

(`make` *type* `.` *args*) *Operation*

Returns a new instance of *type* that has been initialized by sending it an `initialize` message with the extra arguments *args* passed along.

(`%install-method-with-env` *type operation code-body environment*) *Operation*

Adds the specified method to the search table of *type*. It returns *operation*, since this is what some instances of `add-method` are compiled into. Methods that don't close over anything can refer to `%empty-environment`, whose value is an environment object whose vector portion has length zero. It takes care of instance variable mapping conflicts.

(`initialize` *object*) *Operation*

Returns *object*. This no-op is what is shadowed when you define `initialize` methods for new types. (`initialize` *type supertype-list ivar-list*) does the work involved in making a new type. The list of supertypes is used to make a list of all ancestors that is searched at run time to find methods for operations. The ancestor tree is considered to be ordered from bottom to top and from left to right while constructing this list, and duplicates are removed. An error is generated if more than one top-wired type is found in the resulting ancestor list. The instance-variable map of the type is created, with any top-wired type appearing at the beginning, and `variable-length-mixin` appearing at the end if it is present. Any method you define to handle an `initialize` message should return `self`.

`(dynamic-wind` *before-op main-op after-op*`)` *Operation*

Calls the operation *before-op*, calls the operation *main-op*, calls the operation *after-op*, and returns the value returned by *main-op*. If *main-op* is exited abnormally, *after-op* is called automatically on the way out. Similarly, if *main-op* is entered abnormally, *before-op* is called automatically on the way in.

`(call-with-current-continuation` *operation*`)` *Operation*

Calls *operation* with one argument, the current continuation. The synonym `call/cc` is provided for those who feel that `call-with-current-continuation` is excessively verbose.

Chapter 3

Internal Data Format

This chapter describes how memory and tags are set up, and how this implements the object semantics of the language.

3.1 Tag Types

In an effort to reduce the complexity of the bytecode interpreter and to simplify the system in general, there are only four tag types. Tags are stored in the two low order bits of each reference thus simplifying tag manipulation, particularly in the presence of indexed addressing modes.

31 30 29 28 27 26 . . . 11 10 9 8	*7 6 5 4 3 2*	*1 0*	*type*
twos 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

This tagging scheme, along with our object format, does not allow for *arbitrarily* scannable heaps (in which the divisions between objects can be figured out starting the scan at any point in the heap.) In fact, if solitary cells are permitted, as they are in our implementation, scanning the heap starting at the begining is not even possible. However, our garbage collector never needs to scan the heap in such a fashion. Note that there is no extra "gc" bit in every word, but again, our garbage collector requires no such bit.

3.2 Other Immediate Types

References with a tag of $\boxed{1\ 0}$ use the next six bits to specify a subtype.

31 . . . 24	*23 . . . 16*	*15 . . . 8*	*7 . . . 2*	*1 0*	*type*
reserved	font	ascii code	0 0 0 0 0 0	1 0	character

Character is currently the only "other immediate type." More may be added later, in particular Macintosh handles. (At one time weak pointers were represented as their own immediate type, but they are now represented using integers for compatibility with the Scheme standard [3].)

3.3 Memory Structure

Memory is a linear array of *cells*, 32-bit aligned words. These cells are divided into two contiguous chunks: free cells and allocated cells. 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 *aggregate 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 is stores the length of the object, including the first two cells. The remainder of the cells of hold the instance variables. Solitary cells are cells that are not part of any object, but are the targets of locatives. Solitary cells are used heavily in the implementation of variables.

A reference to an object consists of a pointer to that object with a tag of *boxed-object*. References to solitary cells are locatives. Furthermore, locatives may reference cells that are the instance variables of objects. If such an object is ever deallocated by the garbage collector, all of the cells making up the object are made free *except* for those cells that are referenced by locatives, which are not deallocated. These become solitary cells.

3.4 Representation of Specific Types

Consider an object of type *foo*, which is based on *bar* and *baz*. *Bar* had instance variables `bar-1` and `bar-2`, baz has instance variables `baz-1`, `baz-2` and `baz-3`, and *foo* has instance variable `foo-1`. *Foo* inherits the instance variables of the types it is based on, but methods defined for type *foo* can not refer to these inherited variables.

Each type's local instance variables are stored contiguously, and in order of lexical definition, in instances of that type, and of types that inherit it; this allows variable reference to instance variables to be resolved into offsets from the start of the relevent instance variable frame at compile time. Here is an instance of *foo* as it might actually be stored in memory:

reference to type *foo*
value of `foo-1`
value of `baz-1`
value of `baz-2`
value of `baz-3`
value of `bar-1`
value of `bar-2`

Observe that instances of type *foo* are divided into contiguous chunks of instance variables, each inherited from a different supertype. When a type inherits another type through two different routes, it still only inherits the instance variables once.[1] Furthermore, if the instance variables of

[1]This aspect of the language is in flux, and should not be relied upon by users.

two types inherited by a third have the same names they are still distinct instance variables.[2] These semantics allow us to reference instance variables very quickly, once the local instance variable block has been located. It also allows us to use the same compiled code for a single method regardless of whether it is being invoked upon an instance of the type it was added to or on an instance of an inheriting type.

3.5 System Types

This section describes the format of various objects that are directly referenced by the microcode,[3] such as code vectors and catch tags.

It should be emphasized that these system objects are full-fledged objects. They have types which can be inherited and have their methods overridden, just like any other object. The only "magic" thing about these types is that their local instance variables (ie. the system ones) must live at the top of their memory representation, even when inherited. This allows the microcode to locate the values it needs without going through the type heirarchy.

The only constraint this places on the user is that a type may not inherit two types both of which are *top-wired*, for obvious reasons. For example, it is impossible to make a type whose instances are both operations and types.

3.5.1 Methods

A method has two instance variables which hold the code object containing the code that implements the method and the environment vector that holds references to variables that were closed over.[4]

3.5.2 Environment Vectors

Environment vectors have 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. Currently the compiler does not generate such sophisticated constructs.

---

[2] This is in marked contrast to ZetaLisp flavors–that's why variable references in flavors go 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 in [4].

[3] Our microcode is C.

[4] Well, not all closed over variables. Only ones above the locale level. Locale variable references are implemented as inline references to value cells.

3.5.3 Code Vectors

Code lives in 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 existance, treating code vectors like any other vector. Bytecodes are 16 bits long, with the low 2 bits always 0. Here is an example of some stuff taken from the middle of a code vector.

<table>
<tr><td colspan="6">⋮</td></tr>
<tr><td>8 bit inline arg</td><td>6 bit opcode</td><td>0 0</td><td>8 bit inline arg</td><td>6 bit opcode</td><td>0 0</td></tr>
<tr><td colspan="2">14 bit instruction</td><td>0 0</td><td>8 bit inline arg</td><td>6 bit opcode</td><td>0 0</td></tr>
<tr><td colspan="2">14 bit relative address</td><td>0 0</td><td>8 bit inline arg</td><td>6 bit opcode</td><td>0 0</td></tr>
<tr><td>8 bit inline arg</td><td>6 bit opcode</td><td>0 0</td><td>8 bit inline arg</td><td>6 bit opcode</td><td>0 0</td></tr>
<tr><td colspan="2">14 bit instruction</td><td>0 0</td><td colspan="2">14 bit instruction</td><td>0 0</td></tr>
<tr><td colspan="6">arbitrary reference used by last instruction of previous word</td></tr>
<tr><td colspan="2">14 bit instruction</td><td>0 0</td><td>8 bit inline arg</td><td>6 bit opcode</td><td>0 0</td></tr>
<tr><td colspan="6">⋮</td></tr>
</table>

Note the arbitrary reference right in the middle of 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 pc is not currently pointing to an aligned location then the pc 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.

An alternative that was used earlier in the design process was to mandate that all instructions requiring inline arguments occur in a position where the following reference can be fetched without realigning the pc. This requires sometimes inserting a padding `noop` before an instruction that requires an inline argument, and analysis showed that the time required to process a `noop` instruction is much greater than the time required to check if the low bit of a register is on and increment that register if so.

3.5.4 Endianity

The logical order of the instructions in a code vector depends on the endianity of the CPU running the emulator. If the machine is big endian, ie. addresses start at the most significant and of a word and go down (eg. a 68000 or an IBM 370) then instructions are executed left to right in the picture above. Conversely, on a littleendian machine (eg. 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.

`(%big-endian?)` *Operation*

This returns the endianity of the machine that Oaklisp is running on. Endianity is determined by the order in which instructions are fetched, in other words, the order of two 16-bit words within a 32-bit word. This returns true if the first instruction fetched is from the more significant half.

3.5.5 Stack Implementation

Although the value and context stacks are logically contiguous, they are sometimes physically discontinuous. The instructions all assume that stacks live in a designated chunk of memory called the stack buffer. They check if they are about to overflow or enderflow 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* which is allocated on the heap. These overflown 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 [3] is implemented in terms of *stack photos,* which are snapshots of the current state of the two stacks, and which can be restored in the future. A `fill-continuation` instruction forces the stack buffers to be flushed and then copies references to the linked lists of overflow segments into a continuation object.

Actually, in the above treatment we have oversimplified the concept of flushing a stack buffer. 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 `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 that of using 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 did not use a stack buffer but instead implemented stacks as linked lists of segments which always lived in the heap. When pushing over the top of a segment, a couple references were copied from the top of that segment onto a newly allocated segment, providing sufficient hysteresis to prevent repeated pushing and poping along a segment boundary from incurring inordinate overhead. Regretably, substantial storage is wasted by the hysteresis and the overflow and underflow limits vary dynamically wereas in the new system these limits are C link-time constants. Presumably due to these factors, in spite of its old world charm, timing experiments between the old system and the new system were definitive.

3.5.6 Escape Objects

In our implementation of Oaklisp we provide two different escape facilities: `call/cc` and `catch`. The `call/cc` construct is that described in the Scheme standard [3]. The `catch` facility provides with 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 chopped off to the appropriate heights. The slot `saved-wind-count` is used for unwind protection and `saved-fluid-binding-list` is used for fluid variables. Details are given in sections 7.3 and 7.2.

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, 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 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.

3.5.7 Types

Type objects are used when tracing up the type heirarchy in order to find appropriate methods and bp offsets. Since the types are used to find methods, they must be system objects so that reference to their instance variables can be done without sending them explicit messages. The `operation-method-alist` maps from operations to methods handled by the type itself, not any supertype. The `type-bp-alist` maps from types to offsets which are where the appropriate frame of instance variables may be found. The microengine uses a simple move-to-front heuristic in an attempt to reduce the overhead of searching these alists. The `supertype-list` contains a list of the immediate supertypes. Supertypes by inheritance that have instance variables are present in `type-bp-alist`, however.

This is a picture of the `cons-pair` type, as it actually appears in memory:

type	
instance-length:	3
variable-length?:	`#f`
supertype-list:	(*pair* *object*)
ivar-list:	`(the-car the-cdr)`
ivar-count:	2
type-bp-alist:	((*cons-pair* . 1))
operation-method-alist:	((*car* . *meth*) ...)
top-wired?:	`#f`

3.6 Storage Reclamation

Our garbage collector [1] is a variant of Baker's algorithm, a so-called "stop and copy" collector. 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. Due to locatives, the collector makes an extra pass over the data; a paper with more complete details on this latter complication is in press. The weak pointer table is scanned at the end of garbage collection, and references to deallocated objects are discarded.

The user interface to the garbage collector is quite simple. Normally, the user need not be concerned with storage reclamation; upon the exhaustion of storage, the garbage collector is automatically invoked. When this happens some messages are printed; these messages can be supressed with the `-Q` switch. The default size of new space is 1Mb, or 256k references. This can be altered with the `-h` *size* switch, where *size* is measured in bytes. The operations `%gc` and `%full-gc` invoke the garbage collector explicitly. Programs that use weak pointers can be effected by garbage collection; for details, see section 4.3.

The `-G` switch indicates that if and when the world is dumped, and if Oaklisp terminates with an exit code of zero, a full garbage collection should be performed. In full garbage collections preceding world dumps, the root set does not include the stacks.

New space is resized dynamically, being expanded to `RECLAIM_FRACTION` times the amount of unreclaimed data if the fraction of unreclaimed data is above more than one `RECLAIM_FRACTION`’th of new space after a normal garbage collection, or by the minimal amount needed if there is insufficient space available in new space to fulfill the allocation request that triggered the collector. Currently `RECLAIM_FRACTION` is two. The `next_newspace_size` register says how big the next new space allocated will be, and is accessible to Oaklisp code. Its value should not be lowered casually, as the garbage collector will fail if new space is too small to hold all of the non-reclaimed storage from old space. A full garbage collection sets the size of new space back to the value originally specified by the user when Oaklisp was invoked, or the default value if none was specified.

Chapter 4

Stack Machine Architecture

4.1 Registers in the Emulator

This section describes the registers that make up the state of the bytecode emulator, called the processor below.

`pc`: The program counter points to a half reference address, and can not be accessed by register instructions.

`val_stk`: The top of the value stack. Can not be accessed by register instructions.

`cxt_stk`: The top of the context stack. Can not be accessed by register instructions.

`bp`: The base pointer points to the base of the instance variable frame of the current object.

`env`: The current environment object is indexed into to find locatives to lexically closed variables.

`current_method`: The method whose code is currently being executed. This is maintained solely to simplify garbage collection and debugging.

`nargs`: The number of args register is set before a function call and checked as the first action within each function.

`t`: Holds the cannonical truth object, `#t`.

`nil`: Holds the cannonical false object, `#f`, which is also used as the empty list, `()`.

`fixnum_type`: Holds the type of objects with a tag of fixnum.

`loc_type`: Holds the type of objects with a tag of locative.

`subtype_table`: Holds a table of the types of all the immediate subtypes. Currently only the first entry is used.

`cons_type`: Holds the *cons-pair* type, the type of simple conses which are directly manipulated by the processor.

`env_type`: Holds that type of environment vectors, used when making new environment objects.

`object_type`: Holds the type *object* which is at the root of the type hierarchy. Used when calling an operation with no parameters. This should not be necessary in the next version.

`segment_type`: Holds the type of stack segments, for use when the stack is being copied into the heap.

`argless_tag_trap_table`: Holds a table of operations to be called when various instructions fail.

`arged_tag_trap_table`: Holds a table of operations to be called when various instructions fail.

`boot_code`: Holds the method to be called first thing at boot time.

`uninitialized`: Holds the value that gets stuck into newly allocated storage.

`free_point`: Holds the point at which the next heap object will be allocated. Not accessed directly by even the most internal Oaklisp code, as the processor takes care of initialization and gc itself.

`new.end`: Holds the point at which we've run out of storage. An attempt to allocate past here necessitates a garbage collection. Not directly accessed by even the most internal Oaklisp code.

`next_newspace_size`: Holds the size in references of the next new space to be allocated by the garbage collector. This is dynamically adjusted by the garbage collector, so there is usually no need for it to be modified from the Oaklisp level.

4.2 Instruction Set

The instructions follow the same argument order conventions as the language itself. For example, `(store-loc` *loc ref* `)` expects to get *loc* on the top of the value stack and *ref* below it. The instruction format

8 bits	6 bits	2 bits
inline argument	opcode	0 0

leaves eight bits for an inline argument. Instructions that do not require any inline argument actually have "argless instruction" in their instruction field and use the argument field to code for the actual instruction.

Some instructions, eg. `load-imm`, require a complete arbitrary reference as an inline argument. This in incorporated, aligned, directly in the instruction stream. See section 3.5.3 for details. Other instructions, in particular the long branches, require more than an eight bit inline argument but do not need an entire reference. These instructions get a 14 bit inline argument by using the space where the next instruction would normally go, with the last two bits set to zero in case the argument ends up in the low half of a word.

Arithmetic

instruction	*inline arg*	*initial stack*	*final stack*	*extra cell args*
`plus`		2 (fix,fix)	1 (fix)	
`minus`		1 (fix)	1 (fix)	
`subtract`		2 (fix,fix)	1 (fix)	
`times`		2 (fix,fix)	1 (fix)	
`mod`		2 (fix,fix)	1 (fix)	
`div`		2 (fix,fix)	1 (fix)	
`log-op`	n (4 bits)	2 (fix,fix)	1 (fix)	
`bit-not`		1 (fix)	1 (fix)	
`rot`		2 (fix,fix)	1 (fix)	
`ash`		2 (fix,fix)	1 (fix)	

Predicates

instruction	*inline arg*	*initial stack*	*final stack*	*extra cell args*
`eq?`		2 (ref,ref)	1 (bool)	
`not`		1 (ref)	1 (bool)	
`<0?`		1 (fix)	1 (bool)	
`=0?`		1 (fix)	1 (bool)	
`=`		2 (fix,fix)	1 (bool)	
`<`		2 (fix,fix)	1 (bool)	

Control

instruction	*inline arg*	*initial stack*	*final stack*	*extra cell args*
`branch`	rel-addr			
`branch-nil`	rel-addr	1 (ref)		
`branch-t`	rel-addr	1 (ref)		
`long-branch`				0.5
`long-branch-nil`	rel-addr	1 (ref)		0.5
`long-branch-t`	rel-addr	1 (ref)		0.5
`return`				

`catch` and `call/cc` Related

instruction	*inline arg*	*initial stack*	*final stack*	*extra cell args*
`filltag`		1 (tag)	1 (tag)	
`throw`		2 (tag,ref)	1 (ref)	
`fill-continuation`		1 (photo)	1 (photo)	
`continue`		2 (photo,ref)	1 (ref)	

Stack Manipulation

instruction	*inline arg*	*initial stack*	*final stack*	*extra cell args*
All stack references are zero-based. `(swap 0)` is a noop. `(blast` n`)` $\equiv$ `(store-stack` n`) (pop 1)`.				
`pop`	n	n (refs)		
`swap`	n	n (refs)	n (refs)	
`blast`	n	n (refs)	n-1 (refs)	
`blt-stack`	n,m	n+m (refs)	n (refs)	
8 bit ref splits to 4-bit n and m, which are $1 \ldots 16$.				

Register Manipulation

instruction	*inline arg*	*initial stack*	*final stack*	*extra cell args*
`store-reg`	register	1 (ref)	1 (ref)	
`load-reg`	register		1 (ref)	

Addressing Modes

instruction	*inline arg*	*initial stack*	*final stack*	*extra cell args*
`store-env`	offset	1 (ref)	1 (ref)	
`store-stack`	offset	1 (ref)	1 (ref)	
`store-bp`	offset	1 (ref)	1 (ref)	
`store-bp-i`		2 (fix,ref)	1 (ref)	
`contents`		1 (loc)	1 (ref)	
`set-contents`		2 (loc,ref)	1 (ref)	
The next two instructions are the same.				
`load-glo`			1 (ref)	1 (ref)
`load-imm`			1 (ref)	1 (ref)
`load-imm-fix`	n		1 (fix)	
`load-env`	offset		1 (ref)	
`load-stack`	offset		1 (ref)	
`load-bp`	offset		1 (ref)	
`load-bp-i`		1 (fix)	1 (ref)	
Make a locative to the location $offset$ in beyond the `bp` register:				
`make-bp-loc`	offset		1 (loc)	
`locate-bp-i`		1 (fix)	1 (loc)	

Memory Model and Tag Cleaving

instruction	*inline arg*	*initial stack*	*final stack*	*extra cell args*
`get-tag`		1 (ref)	1 (fix)	
`get-data`		1 (ref)	1 (fix)	
`crunch`		2 (fix,fix:tag)	1 (ref)	
`load-type`		1 (ref)	1 (ref:type)	
`load-length`		1 (ref)	1 (fix)	
The next two instructions are not currently used.				
`peek`		1 (fix)	1 (fix:16-bit)	
`poke`		2 (fix,fix:16-bit)	1 (fix:16-bit)	

Misc

instruction	*inline arg*	*initial stack*	*final stack*	*extra cell args*
`check-nargs`	n	1 (op)		
`check-nargs-gte`	n	1 (op)		
`store-nargs`	n			
`noop`				
`allocate`		2 (typ,len)	1 (ref)	
`vlen-allocate`		2 (typ,len)	1 (ref)	
`funcall-tail`		2 (op,obj)	1 (op,obj)	
`funcall-cxt-br`	rel-addr	2 (op,obj)	1 (op,obj)	
`push-cxt`	rel-addr			
`push-cxt-long`				0.5
`big-endian?`			1 (bool)	
`object-hash`		1 (ref)	1 (fix)	
`object-unhash`		1 (fix)	1 (ref)	
`gc`			1 (ref)	
`full-gc`			1 (ref)	
`inc-loc`		2 (loc,fix)	1 (loc)	

List related instructions

instruction	*inline arg*	*initial stack*	*final stack*	*extra cell args*
`cons`		2 (ref,ref)	1 (ref)	
`reverse-cons`		2 (ref,ref)	1 (ref)	
`car`		1 (pair)	1 (ref)	
`cdr`		1 (pair)	1 (ref)	
`set-car`		2 (pair,ref)	1 (ref)	
`set-cdr`		2 (pair,ref)	1 (ref)	
`locate-car`		1 (pair)	1 (loc)	
`locate-cdr`		1 (pair)	1 (loc)	
`assq`		2 (ref,alist)	1 (ref:pair/nil)	

4.3 Weak Pointers

Weak pointers allow users to maintain tenuous references to objects, in the following sense. Let α be a weak pointer to the object *foo*, found by executing the code `(object-hash` *foo*`)`. This α can be dereferenced to yield a normal reference, `(object-unhash` α`)` $\Rightarrow$ *foo*. However, if there is no other way to get a reference to *foo* then the system is free to invalidate α, so `(object-unhash` α`)` $\Rightarrow$ `#f`. In practice, when the garbage collector sees that there are no references to *foo* except for weak pointers it reclaims *foo* and invalidates any weak pointers to it.

Weak pointers are implemented directly by bytecodes because the emulator handles all details of storage allocation and reclamation directly. Weak pointers are represented by integers. Each time call call the `object-hash` is made the argument is looked up in the *weak pointer hash table*. If no entry is found, a counter is incremented and the value of that counter is returned. An entry is made in the *weak pointer table* at an index corresponding to the current value of the counter, so that the weak pointer can be used to get back the original reference, and an entry is make in the weak pointer hash table to ensure that if the weak pointer to the same object is requested twice, the same number will be returned both times. After a garbage collection the weak pointer table is scanned and entries to objects which have been reclaimed are discarded, the weak pointer hash table is cleared, and the data in the weak pointer table is entered into the weak pointer hash table. Although these algorithms are poor if objects with weak pointers to them are frequently reclaimed, in practice this has not been a problem.

Chapter 5

Stack Discipline

This chapter describes how the stacks are organized at the logical level: how temporaries are allocated, how functions call and return work, how escape objects (used in the implementation of catch and throw) work, and how stack snapshots (used in the implementation of call/cc) work.

5.1 Stack Overview

The Oaklisp bytecode machine has a two-stack architecture. The *value stack* contains arbitrary references and is used for storing temporary variables, passing arguments, and returning results. The *context stack* is used for saving non-variable context when calling subroutines. Only context frames are stored on the context stack. This two stack architecture makes tail recursion particularly fast, and is in large part responsible for the speed of function call in this implementation.

Most of the bytecodes are the usual sort of stack instructions, and use only value stack, for instance `plus` and `(swap` *n*`)`. All arguments are passed on the value stack, and the value stack is *not* divided into frames. Methods consume their arguments, returning when they have replaced their arguments with their result or tail recursing when they have replaced their arguments with the appropriate arguments to the operations they are tail recursing to.

Under the current language definition there is no multiple value return, although the bytecode architecture admits such a construct. There are facilities for variable numbers of arguments, which are described in section 7.9.

5.2 Method Invocation/Return

When a method is to be invoked, the arguments and operation are assembled on the value stack in right to left order, ie. the rightmost argument is pushed first and the operation is pushed last. Let us walk through the invokation of `(f x y z)`, where `f` is on operations which is being passed three arguments. Since we evaluate right to left, first we push `z`, thus:

⋮
`z`

continuing, we push the rest of the arguments and the operation, until the stack is of this form.

⋮
z
y
x
f

A (store-nargs 3) instruction is now executed to place the number of arguments in the nargs register, and one of the funcall instructions is executed, which variant depending on whether this is a tail recursive call. If this is not a tail recursive call, the funcall instruction first pushes a frame containing the contents of the current_method, bp and env registers and a return pc onto the context stack. The instruction then examines the top two values, f and x, and looks f up in the operation-method-alist of the type of x, potentially also scanning the supertypes until it finds the appropriate method to be invoked. This method is placed in the current_method register, the method's environment is placed in the env register, the pc is set to the beginning of the method's code block, and the address of the appropriate instance variable frame within x is placed in the bp register. The funcall instruction leaves the value stack and nargs register unchanged:

⋮
z
y
x
f

The first thing the code block of the resultant method executes is one of the check-nargs instructions, in this case perhaps (check-nargs 3). A (check-nargs n) instruction tests if nargs is n, trapping if not. After that, it pops the operation f off the stack. By leaving the operation to be popped off by the check-nargs instruction rather than the funcall instruction, when an an incorrect number of arguments is detected the operation is still available to the error system. The return instruction pops the top frame off the context stack, loads the popped context into the processor, and continues execution. Before a return is executed all of the arguments have been consumed and the result is the only thing left on the stack,

⋮
(f x y z)

5.3 The Context Stack

The only things that can be stored on the context stack are context frames, which each have four values, as below. The push-cxt instruction pushes a context frame onto the context stack. It takes an inline argument, which is the relative address of the desired return point. This allows a context to be pushed whenever convenient, perhaps before the assembly of arguments begins. Earlier in the implementation process there was only one variant of the funcall instruction, which was tail recursive. Non tail recursive calls were compiled as a push-cxt followed by a `funcall-tail`, but because this sequence occured so frequently a combined instruction was implemented to improve code density.

⋮
`pc`
`bp`
`env`
`current_method`
`pc`
`bp`
`env`
`current_method`
`pc`
`bp`
`env`
`current_method`

Actually, the `pc` stored in the context stack is not a raw pointer to the next instruction but rather the offset from the beginning of the current code block, stored as a fixnum. This makes the `return` instruction slightly slower, as the actual return pc must be recomputed, but simplifies the garbage collector. The `bp` is analogously stored with a tag of `locative` so that the garbage collector need not treat it specially. This would cause a problem if the current object were reclaimed and afterwards had one of its instance variables refered to, as all that would be left of the object would be the solitary cell that the saved bp was pointing to, and the rest of the relevent instance variable frame would be gone. This is avoided by having the compiler ensure that a reference to the object in question is retained long enough.

Chapter 6

Methods

In this chapter we describe how methods are created, represented, and looked up. This is intimately related to instance variable reference, so we describe how that works here as well.

6.0.1 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.

  Giving the `-M` switch to a version of the emulator compiled with `FAST` not defined will print an `H` when there is a method cache hit and an `M` when there is a miss. 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 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 average 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.

For most of this tuning we used the time required to compile `compile-bench.oak` as our primary benchmark for determining the speed of generic operations, since the compiler is written in a highly object-oriented style and makes extensive use of inheritance.

6.0.2 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[1] 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

[1]The compiler guesses by attempting to evaluate the type expression at compile time.

created and installed. Creating the method involves pairing the code block with an appropriate environment vector containing references to variables that have been closed over. 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 occations. All other environment vectors are created by pushing the elements of the environment onto the stack and executing the `make-closed-environment` opcode. Environment vectors are never shared in our current implementation, with the exception of the empty environment.

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 `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`.

(`%install-method` *type operation code-body*) *Operation*

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

(`%install-lambda-with-env` *code-body environment*) *Operation*

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

but more efficient.

(`%install-lambda` *code-body*) *Operation*

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

but more efficient.

Chapter 7

Oaklisp Level Implementation

Once the core of the language is up, the rest of the language is implemented using the language core. Some of these new language constructs require some support from the bytecode emulator along with considerable Oaklisp level support. These include such features as `call/cc` and its simple cousin `catch`. Others are implemented entirely in the core language without the use of special purpose bytecodes; in this latter class fall things like infinite precision integers (so called *bignums*), fluid variables, and the error system.

In this chapter we describe the implementation of these constructs, albeit sketchily. For more details, the source code is publicly available. We do not describe the implementation of locales or other extremely high level features; read the source for the details, which are quite straightforward.

7.1 Fluid Variables

Our implementation of fluid variables uses deep binding. A shallow bound or hybrid technology would presumably speed fluid variable reference considerably, but they are used rarely enough that we have not bothered with such optimizations. In addition, shallow binding interacts poorly with multiprocessing.

`fluid-binding-list` *Global Variable*

Hold an association list which associates fluid variables to their values. The `bind` construct simply pushes variable/value pairs onto this list before executing its body and pops them off afterwards.

It would be easy to implement fluid variables using the unwind protection facilities, but instead the abnormal control constructs (`native-catch` and `call/cc`) are careful to save and restore `fluid-binding-list` properly. This avoids the overhead of using the wind facilities and makes sure that (ignoring `wind-protect`) `fluid-binding-list` is only manipulated once for every abnormal exit, no matter how many `bind` constructs are exited and entered along the way.

(`%fluid` *symbol*) *Locatable Operation*

This looks *symbol* up on `fluid-binding-list`. If it is not found an error is signaled. In contrast, `(setter %fluid)` silently adds new fluid variables to the end of the association list, thus creating new top level fluid bindings.

7.2 Unwind Protection

In the presence of `call/cc`, a simple `unwind-protect` construct a. la. Common Lisp does not suffice. Because control can enter a dynamic context which has previously been exited, symmetry requires that if we have forms that get executed automatically when a context is abnormally exited, we must also have ones that get executed automatically when a context is abnormally entered. For this purpose the system maintains some global variables that reflect the state of the current dynamic context with respect to these automatic actions.

`%windings` *Global Variable*

This is a list of wind/unwind action pairs, one of which is pushed on each time we enter a `dynamic-wind` and poped off when we leave it. The wind/unwind action pairs are of the form `(`*after before* `.` *saved-fluid-binding-list*`)` where *before* and *after* are operations, guards to be called when leaving and entering this dynamic context respectively, and *saved-fluid-binding-list* is the appropriate value for `fluid-binding-list` when calling these guard operations.

`%wind-count` *Global Variable*

To reduce `find-join-point`'s complexity from quadratic to linear, we maintain `%wind-count` = `(length %windings)`.

7.3 Catch

The format of catch tags is describe in section 3.5.6. The simplest implementation of `native-catch` would have the `native-catch` macro expand into something that executed the appropriate unwind protect actions and restored the fluid binding list before resuming execution. Regretably, the unwind protect actions can themselves potentially `throw`, so the stacks must not be chopped off until after the unwind protect actions have been completed. For this reason the `throw` operation doesn't just call the `throw` instruction, but first performs all the appropriate unwind protect actions. Along with stack heights, the catch tag contains `saved-wind-count`, which is used to compute how many elements of `%windings` must be popped off and called, and `saved-fluid-binding-list`, which is restored immediately before the stacks are actually chopped off.

7.4 Call/CC

The `call/cc` construct is just like `native-catch`, except that the saved stack state isn't just some offsets but is an entire stack photo (see section 3.5.5), and that not only unwinding but also rewinding actions might need to be done. Because the winding actions might `throw`, it is necessary for the unwind actions to be executed in the stack context where the continuation is invoked, and similarly the rewind actions must be executed in the destination stack context.

`%%join-count` *Global Variable*

`%%new-windings` *Global Variable*

`%%new-wind-count` *Global Variable*

`%%cleanup-needed` *Global Variable*

These global are used to pass information about which rewind actions need to be executed by the destination of the continuation, since the normal parameter passing mechanisms are not available. This would have to be done on a per processor basis in a multithreaded implementation.

Continuations contain `saved-windings` and `saved-wind-count` instance variables, which have the values of `%windings` and `%wind-count` at the time the `%call/cc` was entered. Before the continuation is actually invoked and the destination stack photos restored, the highest join point between current and the destination winding lists is found, and all the unwind actions needed to get down to the join point are executed. Then the stack photo is restored, and in the destination context the rewinding actions are done to get up from the join point to the destination point.

7.5 The Error System

The error system is pretty complete, but is actually not only easy to use, but also intuitive and fun, particularly at the user level.

`(error-return` *message* `.` *body*`)` *Macro*

Evaluates *body* in a dynamic context in which a restart handler is available that can force the form to return. The handler is identified by *string* in the list of choices printed out by the debugger. If the handler is invoked by calling `ret` with an argument in addition to the handler number, the `error-return` form returns this additional value; otherwise it returns `#f`. If no error occurs, an `error-return` form yields the value of *body*.

`(error-restart` *message let-clauses* `.` *body*`)` *Macro*

Acts like a `let`, binding the *let-clauses* as you would expect, except that if an error occurs while evaluating *body*, the user is given the option of specifying new values for the variables of the *let-clauses* and starting *body* again. This is implemented with a `native-catch` and some tricky restart handlers that get pushed onto `(fluid restart-handlers)`.

`(fluid restart-handlers)` *Fluid Variable*

A list of actions that the user can invoke from the debugger in order to restart the computation at various places. Not normally manipulated by user code.

`(fluid debug-level)` *Fluid Variable*

The number of recursive debuggers we're inside. Zero for the top level. Not normally manipulated by user code.

`(catch-errors (`*error-type* [*error-lambda* [*non-error-lambda*]]`)` `.` *body*`)` *Macro*

Evaluates *body*. If an error which is a subtype of *error-type* occurs, `#f` is returned, unless *error-lambda* is given, in which case it is called on the error object. If no error occurs then the result of evaluating *body* is returned, unless *non-error-lambda* is provided in which case it is called on the result of the evaluation of *body* within the context of of the error handler, and the resultant value returned.

`(bind-error-handler (`*error-type handler*`) . `*body*`)` *Macro*
This binds a handler to errors which are subtypes of *error-type*. When such an error occurs, an appropriate error object is created and *handler* is applied to it.

`(invoke-debugger `*error*`)` *Operation*
This error handler, when sent to an error object, invokes the debugger.

`(remember-context `*error after-op*`)` *Operation*
Used to make an error remember the context it occured in, so that even after the context has been exited the error can still be proceeded from, or the debugger can be entered back at the error context. This should always be called tail recursively from a handler, and after it stashes away the continuation it calls *after-op* on *error*. Of course, *after-op* should never return.

`(invoke-in-error-context `*error operation*`)` *Operation*
Go back to the context in which *error* occured and invoke *operation* there.

`(report `*error stream*`)` *Operation*
Write a human readable account of the error to *stream*. Controlled studies have shown that error messages can never be too verbose.

`(proceed `*error value*`)` *Operation*
Proceed from *error*, returning *value*. Of course, it is actually the call to `signal` that returns *value*.

`(signal `*error-type*` . `*args*`)` *Operation*
This signals creates an error of type *error-type* with initialization arguments *args*. It then scans down `(fluid error-handlers)` until it finds a type of error which is a supertype of *error-type*, at which point it sends the corresponding handler to the newly minted error object. If the handler returns, that value is returned by the call to `signal`. One day we'll add a way for a handler to refuse to handle an error, in which case the search for an applicable handler will proceed down the list.

`(fluid error-handlers)` *Fluid Variable*
An association list of mapping error types to error handlers. Users should not touch this directly.

Of course, there are a large number of types of errors used by the system. A few of the more useful to know about are:

`general-error` *Type*
The supertype of all errors. Abstract.

`proceedable-error` *Type*

The supertype of all errors that can be recovered from. Abstract.

`fs-error` *Type*

File system error. Abstract. It has all kinds of subtypes for all the different possible file system error conditions.

`error-opening` *Type*

Abstract. Signaled when a file can't be opened for some reason. Proceeding from this kind of error with a string lets you try opening a different file.

`operation-not-found` *Type*

Signaled when an operation is sent to an object that can't handle it. Proceeding from this kind of error will return a value from the failed call.

`nargs-error` *Type*

Signaled when there are an incorrect number of arguments passed to a function. Proceeding from this will return a value from the failed call. Abstract

`nargs-exact-error` *Type*

Signaled when there are an incorrect number of arguments passed to a method that expects a particular number of arguments.

`nargs-gte-error` *Type*

Signaled when there are an insufficient number of arguments passed to a method that can tolerate extra arguments.

`infinite-loop` *Type*

Signaled when an infinite loop is entered. User programs may wish to signal this as well.

`read-error` *Type*

Some kind of reader syntax error. Abstract. There are about fifty million subtypes, corresponding to all the different constructs that can be malformed, and all the different ways in which they can be malformed. We probably went a little overboard with these.

`user-interrupt` *Type*

Oaklisp received a DEL signal. Through a convoluted series of events in which the UNIX trap handler sets the variable `_del_`, which is detected by the bytecode emulator which pretends that a `noop` instruction failed and passes the `nargs` register to the Oaklisp trap handler which salts the old `nargs` away for restoration upon return and signals this error type, the user usually lands in the debugger after typing Control-C.

7.6 Numbers

Small integers (between -2^{29} and $2^{29}-1$ inclusive) are represented as immediates of type `fixnum` and handled directly by microcode. When arithmetic instructions trap out, due to either their arguments not being `fixnums` or to overflow, an Oaklisp operation corresponding to the bytecode is called. Most of these operations are written in terms of other bytecodes, and should never be shadowed. For instance,

```
(add-method (subtract/2 (number) x y)
  (+ x (- y)))
```

defines subtraction in terms of negation and addition. The trap code also handles fixnum overflow, promoting the operands to `bignums` and dispatching appropriately. The only really primitive operations, which must handle all types of numbers, are `<`, `=`, `minus`, `negative?`, `plus/2`, `times/2`, `/`, `/r`, `quotient`, `remainder`, `quotientm` and `modulo`. Whenever a new type of number is defined, methods for all of the above operations should be added for it, unless the new type is not a subtype of `real`, in which case methods wouldn't make sense for `<`, `negative?`, and perhaps `quotient`, `remainder`, `quotientm` and `modulo`.

7.7 Vectors and Strings

Rather than being built into the emulator, vectors are defined entirely within Oaklisp, albeit with some rather low level constructs.

`variable-length-mixin` *Type*

This type provides a variable amount of stuff at the end of its instances. When a type has this mixed in, whether immediately or deep down in the inheritance tree, it always takes an extra initialization argument which says has long the variable length block at the end should be. This is mixed into such system types as `%code-vector`, `stack-segment`, and `%closed-environment`.

In general, `variable-length-mixin` is used at the implementation level only and should never appear in user code. Typically if you think you want a subtype of `variable-length-mixin`, what you really want is an instance variable bound to a vector.

`(%vref` *variable-length-object n*`)` *Locatable Operation*

This is the accessor operation to get at the extra cells of subtypes of `variable-length-mixin`. It is used in the implementation of variable length structures, and in things like `describe` that look at their internals.

`simple-vector` *Type*

This is a subtype of `vector` with `variable-length-mixin` added and an appropriate `nth` method defined.

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. We therefore use the following somewhat wasteful format.

`string` *Type*

This is a subtype of `simple-vector` with the `nth` method shadowed by one that packs three eight bit characters into the low 24 bits of each fixnum, in littleendian order. 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. Below is the string `"Oaklisp Rules!"` as represented in memory.

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

7.8 Symbols

We do not use any of the fancy techniques used by older dialects, like oblists or symbol buckets. Instead, the standard hash table facility is used for the symbol table.

`symbol-table` *Generic Hash Table*

Maps strings to symbols, using `string-hash-key` to compute the hash and `equal?` to compare strings for equality.

`(intern` *string*`)` *Operation*

Returns a symbol with print name *string* by looking it up in the `symbol-table` and making and installing a new symbol if it isn't found. Strings passed to `intern` should never be side effected afterwards or the symbol table could be corrupted.

`(fluid print-escape)` *Fluid Variable*

This flags whether symbols with weird characters in them should be with the weird characters escaped. It also applies to strings.

`(fluid symbol-slashification-style)` *Fluid Variable*

This flag is only relevent if `(fluid print-escape)` is on. With the value `t-compatible` then the empty symbol is printed as `#[symbol ""]` and all other symbols requiring escaping are printed with a `\` character preceding every character of the symbol. With any other value, escaped symbols are delimited by `|` characters and internal characters `\` and `|` are preceded by `\`.

7.9 Variable Numbers of Arguments

The formal parameter list of a method is permitted to be improper, with the terminal atom being a magic token representing the rest of the arguments. The only legal use for this magic token is as the terminal member of an improper argument list of a tail recursive call, and as an argument to the special form `rest-length`. Methods that accept a variable number of arguments must exit tail recursively and must pass along their magic token in their tail recursive call, unless they know that they actually received no extra arguments.

`(rest-length` *varargs-token*`)` *Special Form*

Returns the number of trailing arguments represented by *varargs-token*.

For example, this is legal,

```
(define (okay x y . rest)
  (if (zero? (rest-length rest))
      'nanu-nanu
      (list 'you x y 'sucker . rest)))
```

while the following are not, the first because it has an exit when there might be extra arguments which does not pass the extra arguments along tail recursively, and the second because it tries to pass along the extra arguments in a non tail recursive position.

```
(define (not-okay x y . rest)
  (if (eq? x y)
      'nanu-nanu
      (list 'you x y 'sucker . rest)))

(define (also-bad x y . rest)
  (append (list 'you x 'sucker . rest) y))
```

The implementation behind this is very simple: extra arguments are ignored by the compiler, except that it emits a `check-nargs-gte` in place of a `chech-nargs` at the top of the method code body and does a little computation to figure out what the value to put in the `nargs` register when it sees rest argument at the tail of a call. When all the user wishes to do is pass the extra arguments along in the way that the `make` method passes extra args along to `initialize`, this mechanism is both convenient and efficient. Sometimes the user needs to actually get into the extra arguments though, so some operations are provided to make handling variable numbers of arguments easier.

`(consume-args` *value* `.` *extra*`)` *Operation*

Returns *value*.

`(listify-args` *operation* `.` *args*`)` *Operation*

Calls *operation* with a single argument, a list of *args*.

There is also a macro package that implements optional and keyword arguments using these facilities, and the Scheme compatiblity package redefines `add-method` so that, as required by the Scheme standard [3], extra arguments are made into a list.

Chapter 8

The Compiler

8.0.1 File Types

There are a number of different kinds of object files, distinguished by extension.

extension	*file type*
`.oak`	Oaklisp source file
`.omac`	Macroexpanded Oaklisp source file
`.ou`	Assembly file, not peephole optimized
`.oc`	Assembly file, peephole optimized
`.oa`	Assembled object file

`compiler-from-extension` *Global Variable*
The extension of the input files the compiler will read. Default `".oak"`. This variable is in the compiler locale.

`compiler-to-extension` *Global Variable*
The extension the the output files the compiler will produce. Default `".oa"`. This variable is in the compiler locale.

`compiler-noisiness` *Global Variable*
The amount of noise the compiler should produce; zero for none, 1 for a little, and 2 for a lot. Default value is 1, but the `oakliszt` batch file compiler sets it to zero. This variable is in the compiler locale.

8.0.2 Object File Formats

8.0.3 Compiler Internals

Some compiler internals documentation. Very sketchy, just enough to give people a vague idea of what the internal program representation is and what the various passes are for.

Chapter 9

Bootstrapping

In this chapter we describe new versions of Oaklisp are created. Essentially, the process is quite similar to the way in which a C program is created. First the Oaklisp source files which make up the *cold world load* are compiled to produce object files. Then a linker, originally written in T but now an Oaklisp program, takes these object files and lays them all out in memory, resolving references to global variables and laying out quoted constants refered to in the code. The linker also puts a map of where it allocated various globals and such in memory. At this point, the cold world (named `oaklisp.cold`) is booted, and the files that the linker layed out in memory are thereby executed, sequentially. These files gradually build all the infrastructure required for a full Oaklisp world. The first files are written at an extremely low level, and make things like `make` and `cons` work. Later files bring up more advanced constructs, until finally there is enough for object files to be loaded. At this point the world is dumped to `oaklisp.ol`, and then this world is booted and has files loaded into it using the normal file loading mechanisms until the full Oaklisp world, `oaklisp.olc`, is built.

The formats of these files is very simple. They contain a header which gives the length of the various segments and the values of some registers. This is followed by a memory image, with pointers given as offsets from the beginning of the image. This is followed by the weak pointer table.

The cold world is in a hexidecimal format, with each reference represented as a space followed by a sequence of hexidecimal digits. Carriage returns may optionally preceed spaces. Actually, the space refered to above can be either a space character or the ^ character. The later indicates that the following reference contains bytecodes. Since bytecodes are ordered differently depending on the endianity of the machine, the hex format world loader swaps the two instructions on little endian machines but not on big endian machines. This keeps the cold load file independent of endianity.

The warm world loads are in a binary format and are not independent of endianity. For this reason, warm world extensions start with `.ol` for big endian versions and `.lo` for little endian versions. The emulator replaces the characters `%%` in the command line file argument (or the default world in `config.h`) with either `ol` or `lo`, depending on whether `BIG_ENDIAN` is defined.

To make Oaklisp dump itself upon exiting use the `-d -b` switches when invoking Oaklisp. After Oaklisp has exited, the emulator will prompt for a filename to dump the world image to, unless this filename has been provided with the `-f` *filename* switch. Usually the `-G` switch is also given when the world is being dumped.

Chapter 10

Administrative Details

10.1 Getting a Copy

The most recent released version of Oaklisp, along with the manuals, are available at http://www.cs.unm.edu/~bap/oaklisp/

10.2 Bugs

The following are known serious problems and inadequacies of the current implementation. People are invited to work on remedying them. None of these are fundamental; they're simply due to lack of either effort or motivation.

- Floating point numbers are not supported. Rationals can be used to make up for this lack.
- In contrast to the error handling system, which is Industrial Strength, the debugger barely exists.
- There is no foreign function interface for loading and calling C routines from a running Oaklisp.

Bug reports and enhancements should be sent to `bap@cs.unm.edu`

We appreciate enhancements (especially in the form of patch files), bug fixes, and bug reports. We are particularly grateful for porting problem fixes. In a bug report, please include the precise version of Oaklisp, which is indicated by the date at the end of the tar file. And please try to make sure that it's really a bug and not a feature, and pretty please, if at all possible, find a *very short* program that manifests your bug. In any case please be aware that we are under no obligation to respond to bug reports in any way whatsoever.

10.3 Copyright and Lack of Warranty

The Oaklisp copyright belongs to its authors. It is authorized for distribution under the GNU General Public License, version 2, copies of which are readily obtainable from the Free Software Foundation. There is no warranty; use at your own risk. For more precise information, see the COPYING file in the Oaklisp source distribution.

Bibliography

[1] Barak A. Pearlmutter. Garbage collection with pointers to single cells. *Communications of the ACM*, 39(12):202–206, December 1996.

[2] Barak A. Pearlmutter and Kevin J. Lang. The implementation of Oaklisp. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 189–215. MIT Press, 1991.

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

[4] Alan 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. Special issue of *SIGPLAN Notices*.

Index