LL System Design

Kurt Stephens

Chicago Lisp Users Group
CashNetUSA
kurt@chicagolisp.org

2008/06/20

Introduction

- Motivation
- Language
  - Scheme but not just Scheme
- Implementation
  - Still buggy and slow after 10 years :)
  - Similar to Oaklisp:
    - but has flonum, but no bignum.
    - all method invocations are by C function pointer call.

«Because LL is self-referential, concepts and constructs to be illuminated later will be marked with "☼".»

Why Bother?

- Wanted an embeddable Scheme:
  - with a clean C namespace
  - with proper tail-calls ☼
  - with an object-oriented foundation
  - with introspection
- Platform for ideas:
  - Interpreter multiplicity, self-referential systems
  - Tagging, memory, GC, linkage, compilation, FFI.
- Another plate of hubris? :)

Language

- LL is an pure object-oriented, lexically-scoped Lisp
- Supports multiple inheritance and mixins
- R4RS Scheme compliance is built on object-oriented core
- Based on Oaklisp and Dylan syntax and semantics
- Extensible and embedded in C
- Introspection

Constructs

- Objects → #<<type> <object>>
- Types → #<<type> <type>>
- Operations → #<<type> <operation>>
- Messages → #<<type> <message>>
- Methods → #<<type> <method>>
- Locatives → #<<type> <locative>> ☼

Objects

- Have a type.
- Have a fixed number of slots.
- Immediate objects have no slots:
  - <fixnum>, <flonum>, <locative> ☼
- Are created by sending the **make** operation to a type object with arguments for a type-specific **initialize** method.

<object>

- (get-type 5) ; => <fixnum>
- (get-type 5.5) ; => <flonum>
- (get-type <fixnum>) ; => <type>
- (get-type <type>) ; => <object>
- (make <pair> 1 2) ; => (1 . 2)
- (eq? (make <symbol> "foo") 'foo)
  ; => #t
- (%%slot (make <pair> 1 2) 1) ;;; unsafe! ☼
  ; => 1

Types

- Are subtypes of <object>.
- Are named ***<something>*** by convention.
- Can be anonymous.
- Can be subtyped (meta-types).
- Have a list of supertypes.
- Have a list of slots.
- Have a mapping of operations to methods.
- Core LL types have analogous C structs. ☼

<type>

- (type-supers <object>) ; => ()
- (type-supers <type>) ; => (<object>)
- (type-supers <pair>) ; => (<list> <object>)
- (type-slots <object>) ; => ((isa 0)) ☼
- (type-slots (get-type '(1 2))
  ; => ((car 0) (cdr 4)) ☼
- (define <my-cons>
  (make <type>
  (list <object>) ; supertypes
  (list 'car 'cdr))) ; slots

Mutable .vs. Immutable

- By convention mutable types are subtypes of immutable types.
  - ```
    (define x (cons 1 2))
    (get-type x)          ; => #<<type> <mutable-pair>>
    x                     ; => (1 . 2)
    (set-car! x 'a)
    x                     ; => (a . 2)
    ```
  - ```
    (make-immutable x)
    (get-type x)          ; => #<<type> <pair>>
    (set-car! x ”foo”)    ; => ERROR
    ```

Operations

- Are objects.
- Can be anonymous.
- Can be subtyped.
- Have a mapping of types to methods.
- Contain a method lookup cache. ☼
- Associate the receiver type (the first argument or <object>) and an implementation method.
- Are the only objects that can be “applied”. ☼

<operation>

- (define my-car (make <operation>))
- (define my-cdr (make <operation>))
- (define my-func
    (lambda (x)   ; ☼
        (+ (my-car x) (my-cdr x))))

Settable Operations

- Are operations that have a setter <operation>.
  - car ; => #<locatable-operation car> ☼
  - (type-supers (get-type car))
    ; => (<settable-operation>)
  - (setter car) ; => #<operation set-car!>
  - setter ; => #<locatable-operation setter>
  - (define x (cons 1 2))
    x ; => (1 . 2)
    (car x) ; => 1
    ((setter car) x 'a)
    x ; => (a . 2)

(set! (op . op-args) value)

- Syntax is transformed into the application of the setter of a <settable-operation>:
  - (set! (op . op-args) value) =>
    ((setter op) . op-args value)
- (define x (car 1 2))
  (set! (car x) 'b)
  x ; => (b . 2)
- (define v (make <vector> 4 'a))
  (set! (vector-ref v 2) 'b)
  v ; => #(a a b a)

Messages

- Are objects.
- Are created by the application of an operation to a receiver and some arguments.
- Lookup is based on the <type> of the receiver.
- Without arguments are messages to a non-existent <object>. ☼
- Implementation:
  - Created on a message stack. ☼
  - Arguments and results are on a separate value stack. ☼

Message Lookup

- (cons 1 2) =>
  - Receiver type => <fixnum>
  - Implementation type => <object>
- (newline) => ...
  - Receiver type => <object>
  - Implementation type => <object>
- (+ 1 2) => ...
  - Receiver type => <fixnum>
  - Implementation type => <fixnum>

Methods

- Are objects.
- Are anonymous.
- Can be subtyped.
- Can lexically “close-over” global bindings, formal arguments or object slots. ☼
- Give lexical scope to slots in an object of the method's implementation type. ☼
- Are resolved at run-time by operation and type.

Implemented with a primitive C function pointer. ☼

(add-method ...)

- The primitive closure special form:

  (%method slots formals . body) ☼

  - Handled directly by the compiler.

- (add-method (op (type . slots) . formals) . body)
  ; =>
  (%add-method type op
  (%method slots formals . body))
  - %add-method is defined on <type> and returns op.

<method>

- ```
  (add-method
    (initialize (<my-cons> car cdr) self a d)
      (set! car a)
      (set! cdr d)
      self))
  ```
- ```
  (define my-car (make <settable-operation>))
  (add-method
      (my-car (<my-cons> car) self)
          car)
  (add-method
      ((setter my-car) (<my-cons> car) self value)
          (set! car value))
  ```

Operation Overloading

- (define add (make <operation>))
  (add-method (add (<string>) a b))
    (string-concat a b))
  (add-method (add (<number>) a b))
    (+ a b))

- (add ”foo” ”bar”) ; => ”foobar”
  (add 2 3) ; => 5
  (add ”foo” 3) ; => #<ERROR!>

Lambda

- Lambdas are anonymous operations.
- An anonymous lambda <operation> is implemented by a <method> defined on <object>.
- Lambda operations can be overloaded for other receiver (first argument) types.

(lambda formals . body)

- ```
  (define (foo bar)
    (+ bar 5))
  =>
  (define foo (lambda (bar)
    (+ bar 5))
  ```
- ```
  =>
  (define foo
    (add-method ((make <operation>) (<object>)
      bar)
      (+ bar 5)))
  ```

Locatives

- Locatives are safe, language-level pointers to value locations, such as:
  - arguments, global bindings or object slots.
- Simplify implementation of closures, call/cc, great for mutable object slots.
- <method> environment vectors contain locatives to closed-over formals and slots which have been moved to the heap. ☼

Locative Operations

- (make-locative variable-or-application)
  - creates a locative; closes-over a variable.
- (contents locative)
  - gets the contents of the location.
- (set-contents! locative value)
  - sets the contents <u>of</u> the locative.
- Identity Transforms:
  - (locative-contents locative) => locative ☼
  - (contents (make-locative x)) => x
  - (make-locative (contents x)) => x

<locative>

- ```
  (define x 5)
  (define l (make-locative x))
  x                          ; => 5
  (contents l)               ; => 5
  (set-contents! l 10)
  (contents l)               ; => 10
  x                          ; => 10
  ```
- ```
  (define x (cons 1 2))
  (define l ((locater car) x))
  (set-contents! l 'foo)
  x                          ; => (foo . 2)
  ```

(set! (contents . args) x)

- (set! (contents locative) value) =>
  ((setter contents) locative value) =>
  (set-contents! locative value)
  - ```
    (define x (cons 1 2))
    (define l (make-locative (car x)))
    (set! (contents l) 'a)
    x                                    ; => (a . 2)
    ```
    ```

Locatable Operation

- A <locatable-operation> is a <settable-operation> that has a locater <operation>.
- (locator op) returns an <operation> that returns a <locative> to a value:
  - car ; => #<locatable-operation car>
  - (setter car) ; => #<operation set-car!>
  - (locater car) ; => #<operation locative-car>
  - ```
    (define x (cons 1 2))
    (define f (locater car))
    (f x)              ; => #<locative (car x)>
    (contents (f x))   ; => 1
    ```
    ```

(make-locative (op . args))

- (make-locative (op . args) =>
  ((locator op) . args)
  - ```
    (define x (cons 1 2))
    (define l (make-locative (car x)))
    x                          ; => (1 . 2)
    (set-contents! l 'a)
    x                          ; => (a . 2)
    ```
- (make-locative x) on a global binding, lexical variable or object slot is a special form. ☼
    ```

Messaging

- Message Object
  - Operation
  - Receiver and Type
  - Arguments
- Method Lookup
  - Method
  - Implementation type
  - Type offset ☼
- Method Application

Message Object

- Message objects have:
  - An operation
  - An argument list (including the receiver in car)
- Method lookup computes:
  - The receiver type
  - The method implementing the operation for the receiver type
  - The type that implemented the method
- Method application:
  - Calls the method with the message.

Mutual Tail Recursion

```
(define (f a)
  (write (cons 'f a)) (newline)
  (g (+ a 1)))                          ; tail call

(define (g b)
  (write (cons 'g b)) (newline)
  (f (* b 2)))                          ; tail call

(f 0)                                   ; normal call
```

Message Protocol

- <message> objects are created.
- Two explicit stacks:
  - <message> stack.
  - Value stack for arguments and results.
- Two stacks simplify tail-calls. ☼
- Both stacks use stack buffers for fast call/cc. ☼
- Do not create locatives to arguments on stack for compatibility with call/cc. ☼
- Avoid passing arguments to lookup primitives, use stack pointers. ☼

Operation Application

- (operation . arguments) =>
  (apply operation arguments)
- Pseudo-code to implement (apply ...) =>
  - (define *value-stack* ())
  - (define *method-stack* ())
  - (define (%call operation . arguments) ...)
  - (define (%call-tail operation . arguments) ...)
  - (define (%return result) ...)

(%call operation . arguments)

```
(define (%call operation . arguments)
  (%push-each! *value-stack* arguments)
  (%push! *message-stack*
    (make <message>
      operation (%length arguments)))
  (while (%lookup-and-apply)
  ;; nothing
 )
  (%pop! *value-stack*))
```

(%lookup-and-apply)

```
(define (%lookup-and-apply)
  (%call
    (%method-function
      (%lookup (%top *message-stack*))))))
```

(%lookup message)

```
(define (%lookup message)
  (let* ((type (%receiver-type message))
         (meth (assq (%operation message)
                     (%op-meth-alist type)))
    (if meth
        (begin
        (%set-method! message meth)
        (%set-method-impl! message type)
        meth
      )
      (%lookup-super message))))
```

(%return result)

- (define (%return result))
  (%pop-arguments!)
  (%pop! *message-stack*)
  (%push! *value-stack* result)
  #f)
- #f tells caller's %apply (while ...) loop to stop.

(%call-tail operation . arguments)

- ```
  (define (%call-tail operation . arguments)
    (let ((message (%top *message-stack*)))
      (%pop-arguments!)
      (%push-each! *value-stack* arguments)
      (%set-operation! message operation)
      (%set-arguments! message
        (make-locative (%top *value-stack*)))
      #t)
  ```
- #t tells caller's %apply (while ...) loop to keep going.
  ```

Interpreter Implementation

- Values, Boxing and Tagging
- Object Layouts
- Runtime Bootstrapping
- Messaging
- Continuations, Catch/Throw
- Evaluation
- Performance

Values, Boxing and Tagging

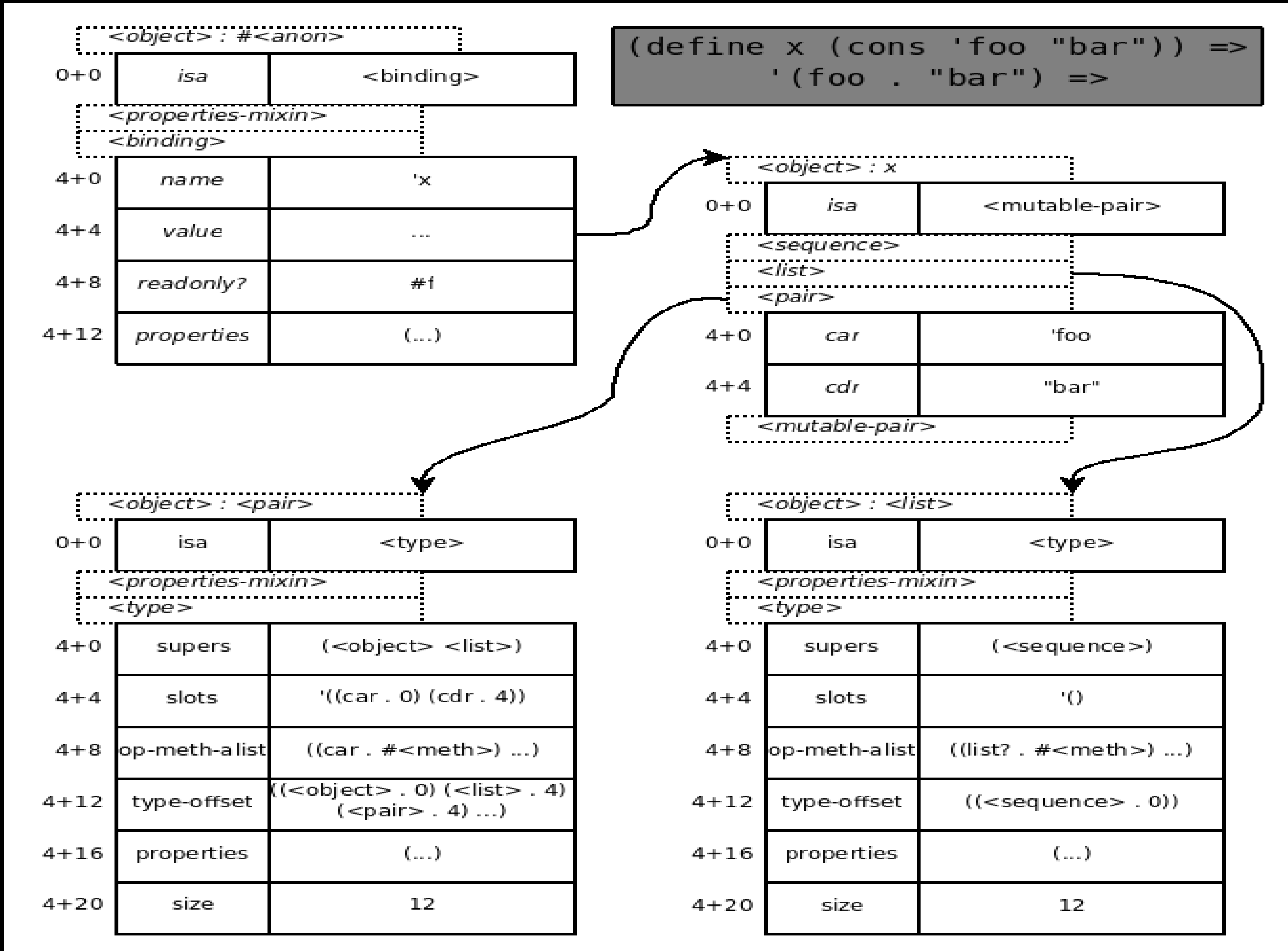
- Values are machine words: *
  - 32-bit words => 4 byte alignment.
  - Defined in C as `typedef unsigned int ll_v`.
- Boxing Values
  - Some objects cannot fit in a machine word.
- Tagging Values
  - Not all values are pointers.
  - Tagging takes up space in machine words.
- Tagging Schemes
  - Some schemes are better than others.

Boxing and Tagging

- Most CPUs do not support latent typing in microcode.
- Some interpreter objects can fit in machine words, many cannot.
- Allocating memory for every object (including integers, floats) is expensive, if not impossible.
- Trade off space, precision, performance and complexity.
- Play nice with C but don't give up the good fight.

Allocated Object Layout

- Each slot is a word. *
- Each ancestor type is given one block of slots.
- Offsets of each ancestor type is stored in each type.
- Mixins have no slots.
- Core types are defined by C macros:
  - Define C structures with inheritance.
  - Create interpreter <type> objects for introspection.



(define x (cons 'foo "bar")) =>
'(foo . "bar") =>
<object> : #<anon>
0+0 isa <binding>
<properties-mixin>
<binding>
4+0 name 'x
4+4 value ...
4+8 readonly? #f
4+12 properties (...)
<object> : x
0+0 isa <mutable-pair>
<sequence>
<list>
<pair>
4+0 car 'foo
4+4 cdr "bar"
<mutable-pair>
<object> : <pair>
0+0 isa <type>
<properties-mixin>
<type>
4+0 supers (<object> <list>)
4+4 slots '((car . 0) (cdr . 4))
4+8 op-meth-alist ((car . #<meth>) ...)
4+12 type-offset ((<object> . 0) (<list> . 4) (<pair> . 4) ...)
4+16 properties (...)
4+20 size 12
<object> : <list>
0+0 isa <type>
<properties-mixin>
<type>
4+0 supers (<sequence>)
4+4 slots '()
4+8 op-meth-alist ((list? . #<meth>) ...)
4+12 type-offset ((<sequence> . 0))
4+16 properties (...)
4+20 size 12

<pair> type definition

- (get-type (cons car cdr)) => <mutable-pair> ☼
  - <pair> C definition
  - <pair> C struct (generated)
  - <pair> LL <type> definition (generated)

<pair> C definition

- ll_define_type(object, type)
      ll_define_type_slot(object, type, ll_v, isa)
  ll_define_type_end(object, type)
- ll_define_type(pair, type)
      ll_define_type_super(pair, type, object)
      ll_define_type_slot(pair, type, ll_v, car)
      ll_define_type_slot(pair, type, ll_v, cdr)
  ll_define_type_end(pair, type)
- ll_define_type(mutable_pair, type)
      ll_define_type_super(pair, type, pair)
  ll_define_type_end(pair, type)

<pair> C struct generated

```
                                              /* type slot blocks */
struct ll_ts_object { ll_v isa; };
struct ll_ts_pair { ll_v car; ll_v cdr; };
struct ll_ts_mutable_pair { /* empty */ };

                                              /* type actual structure */
struct ll_tsa_object {
    ll_v isa;
};
struct ll_tsa_pair {
    struct ll_ts_object    super_object;
    struct ll_ts_pair      super_cons;
};
struct ll_tsa_mutable_pair {
    struct ll_tsa_pair     super_pair;
};
```

<pair> LL definition generated

```
(define <object>
    (make <type>
        '()                   ; supers
        '(isa) ))             ; slots
```

```
(define <pair>
    (make <type>
        (list <object>)       ; supers
        '(car cdr) ))         ; slots
```

```
(define <mutable-pair>
    (make <type>
        (list <pair>)         ; supers
        '() ))                ; slots
```

Tagging

- **Tagging** is a compromise between the **statically-typed** world of the hardware and the **latently-typed** world of the interpreter under the performance constraints of the machine.
- After tagging most native machine object values can fit in machine words; objects that cannot fit in machine words must be allocated.
- Tagging schemes effect:
  - GC, memory layout, code size, FFI, performance.

Tagging Schemes

- Trade off word precision for indirection and storage reduction.
  - High-bit tags
  - Low-bit tags
  - Variable-width tags
  - Fixed-width tags
  - Dedicated tags:
    - Lisp machines.
    - Some experimental HW with dedicated GC support.

High Bit Tagging

- Requires special memory management: mmap(), etc.
- Presumes small type domain and limited object frequency.
- Complicate GC.
- Older Common Lisp implementations.

Low Bit Tagging

- Requires no special memory management.
- Open-ended type domain and object frequency.
- Compatible with off-the-shelf conservative GCs.
- Use tag bits normally unused due to machine alignment.
- Most new language implementations.

ll_TAG(ll_v)

- Tags are low-bit tags: (log2(sizeof(int)))
  - 32-bit words => 2-bit tags; 64 bits => 3-bit tags
- From ll/value.h:
  - ```
    33222222222211111111110000000000
    10987654321098765432109876543210
    --------------------------------
    fixnum                (immediate)|00
    locative                (address)|01
    flonum                (immediate)|10
    boxed object            (address)|11
    ```
    ```

Tagging

- Values converted from statically-typed values (CPU) to latently-typed values (interpreter).
- Lower 2-bit tags for all values:
  - ll_BOX_fixnum(x) =>
    x >> 2 ==
    x * 4
  - ll_UNBOX_fixnum(x) =>
    x << 2 ==
    x / 4
- Some integer word values cannot be tagged, must be dynamically allocated as boxed object: bignums.

Fixnum Tagging/Boxing

- ll_BOX_fixnum(int)
  - ll_BOX_fixnum(0) => 0 << 2 => (ll_v) 0
  - ll_BOX_fixnum(1) => 1 << 2 => (ll_v) 4
  - ll_BOX_fixnum(-5) => -5 << 2 => (ll_v) -20
- ll_UNBOX_fixnum(ll_v)
  - ll_UNBOX_fixnum(0) => (int) 0
  - ll_UNBOX_fixnum(4) => (int) 1
  - ll_UNBOX_fixnum(-20) => (int) -5

Why ll_TAG(fixnum) == 0?

- Fixnum addition, subtraction and vector element offset are common.
- X >> 2 == X * 4
- sizeof(ll_v) == sizeof(int) == 4
- 0 >> 2 == 0 * 4 == 0
- Overflow/underflow still an issue but uncommon in practice.

If ll_TAG(fixnum) != 0

- Scheme: (+ x y) =>
  ```
  C:    ll_BOX_fixnum(
            ll_UNBOX_fixnum(x) +
            ll_UNBOX_fixnum(y))
  ```
- Scheme: (+ x y) =>
  ```
  C:    ll_BOX_fixnum(
            ll_UNBOX_fixnum(x) -
            ll_UNBOX_fixnum(y))
  ```
- Scheme: (vector-ref v i) =>
  ```
  C:    *(ll_v*) (((char*) ll_vector_ptr(v))) +
                        ll_UNBOX_fixnum(i) * 4)
  ```

If ll_TAG(fixnum) == 0

- Scheme: (+ x y) => C: (x + y)
- Scheme: (- x y) => C: (x - y)
- Scheme: (vector-ref v i) => C:
  `*(ll_v*) (((char*) ll_vector_ptr(v)) + i)`
- Scheme: (string-ref s i) => C:
  `*(ll_v*) (((char*) ll_string_ptr(s)) + (i >> 2))`
- Still need to check for underflow/overflow, but checks can be done in a separate pipeline.

Allocated Reference Tagging/Boxing

- First slot in all allocated objects is the object's <type>.
- Lowest 2 bit of 32-bit word addresses are 00 due to machine word alignment.
- Offset allocated object address with ll_TAG_ref:
  - ll_BOX_ref(x) => (((char*) x) + ll_TAG_ref)
  - ll_UNBOX_ref(x) => ((void*)((x) - ll_TAG_ref))
  - ll_SLOTS_ref(x) => ((ll_v*)ll_UNBOX_ref(x))
  - ll_TYPE_ref(x) => ll_SLOTS_ref(x)[0]

Why ll_TAG_ref == 3

- Most slot access will be an offset from C pointer to allocated structure.
- C: ((type*) ptr)->slot =>
  (typeof(type, slot))(ptr + offsetof(type, slot))
- Most slot accesses are unaffected by **subtracting** tag bits after C compiler optimizations.
- Getting the object's type (object slot 0) is more complex at the expense of making <fixnum> tagging faster.

If ll_TAG_ref == 0

- (car x) =>
  - *(ll_v*)(ll_UNBOX_ref(x) +
      offsetof(ll_ts_pair, car)) =>
    (ll_v)(x + 0 + 4) =>
    (ll_v)(x + 4)
  - Non-zero offset has cost.
- (get-type x) =>
  - *(ll_v*)(ll_UNBOX_ref(x) +
      offsetof(ll_ts_object, isa)) =>
    (ll_v)(x + 0 + 0) =>
    (ll_v)(x)
  - Zero offset has no cost.

If ll_TAG_ref == 3

- (car x) =>
  - *(ll_v*)(ll_UNBOX_ref(x) +
        offsetof(ll_ts_pair, car)) =>
    (ll_v)(x - 3 + 4) =>
    (ll_v)(x + 1)
  - Tag removal has no additional cost with offset.
- (get-type x) =>
  - *(ll_v*)(ll_UNBOX_ref(x) +
        offsetof(ll_ts_object, isa)) =>
    (ll_v)(x - 3 + 0) =>
    (ll_v)(x - 3)
  - Tag removal and offset has a unified cost.

ll_TAG(flonum)

- Trade lower 2-bits of C float mantissa precision for tag bits instead of allocating objects.
- C inline functions ll_BOX_float(float) and ll_UNBOX_float(ll_v) use a i386 C union.
- Some C float values cannot be represented.
- Reasonable trade-off since operations on floats are often inexact anyway.
- Does not prevent creation of full precisions float or double boxed values using a new <type>.

Runtime Bootstrapping

- The runtime itself is accessible by the user as a <%runtime> object.
- Some objects are statically allocated in the ll_tsa__runtime C structure:
  - Symbols
  - Global bindings
  - Character objects, nil, #f, #t.
- nil != ll_BOX_ref(0)
- The runtime system can be modified by itself.

Bootstrapping

- Because LL is self-referential, bootstrapping requires very controlled object allocation and initialization:
- Type layout and initialization is implemented in C **and** LL.
- At a certain point, the core constants, types, bindings, operations and methods are complete enough allow ll_send() to be usable for the remainder the initialization.
- LL_DEBUG_INIT=1 ./llt

nil != II_BOX_ref(0)

- If nil was a boxed reference to NULL, the entire system would need NULL checks.
- There is a single object instance of the <null> type.
- The nil symbol has a read-only global binding and is specially recognized by the compiler.
- This complicates bootstrapping the runtime but this simplifies the system by not introducing special cases in the very lowest levels of the message protocol.

Messaging in C

- C macros
  - Normal calls:
    ll_v ll_call(op, _N(arg1, ... argN))
  - Tail calls:
    void ll_call_tail(op, _N(arg1, ... argN))
  - Return from call:
    void ll_return(val)
  - Manipulate stacks inline.
  - Allocate and use inline lookup caches. ☼

Tail-calls in C

- ll_call() puts a new <message> on the message stack and loops while the return value of method function is true (non-zero).
- ll_call_tail() removes current arguments from value stack and replaces current <message> with new message operation and arguments and returns true (non-zero).
- ll_return() pops the current arguments, pushes the result onto the value stack and returns 0.

Mutual Tail Recursion : C

- /*pseudo-C: vs is the value stack, ms is the method stack*/

```
int f( /* a = vs.top() */ ) { /* (g (+ a 1)) */
     ll_v temp = vs.pop() + ll_BOX_fixnum(1);
     vs.push(temp); ms.top().operation = ll_o(g);
    return 1;
}
int g( /* b = vs.top() */ ) { /* (f (* b 2)) */
     ll_v temp = vs.pop() / 4 * ll_BOX_fixnum(2);
     vs.push(temp); ms.top().operation = ll_o(f);
     return 1;
}
int temp_func() {
  vs.push(0); ms.push(f); /* f(0) */
  while ( ms.top().lookup()->func() ) ;
}
```

Method Lookup Performance

- Lambda operation
- Operation lookup cache
- Call site lookup cache
- Move-to-front Heuristic
- Supertypes lookup

Lambda Operations

- Each <operation> object has a lambda cache for operations with methods defined on only one <type>.
  - <object> is a pinned global and its operation-method association lists is also pinned.
  - This allows lambdas to be GCed if the anonymous <operation> is no longer accessible.
- The lambda <method> is moved into <object> if a <method> on another <type> is added.

Operation lookup cache

- Each <operation> object has a single-entry lookup cache.
  - Cache is invalidated when a <method> is added or removed from the <operation>.
- Each <operation> object has a version number.
  - Version number is incremented when a <method> is added or removed from the <operation>.

Call site lookup cache

- Each call site has a multi-entry lookup cache table.
  - Each entry keeps track of the operation version at time of cache fill.
  - If the operation version has changed, the entry is refilled.
  - Most-popular entries are moved to the front of the table.
  - Caveats: inaccessible operations may be stuck in caches.

Move-to-front Heuristic

- The <type>'s op-meth-alist is scanned if the previous caches are not filled.
- The matched association of the op-meth-alist is moved to the front of the list.
- Commonly-applied methods are at the front of the list.
- Consider making op-meth associations weak.

Supertype Lookup

- Supertype lookup routines:
  - Are recursive.
  - Do not pass the <message> object.
  - Return 1 if match is found.
  - Return 0 if more searching is required.
  - Could be changed to use an explicit stack.

Catch/Throw

- Catch/Throw
  - Lighter than call/cc
  - ll_CATCH(type)
    - C setjmp()
    - Creates an anonymous <operation> with a <method> that references the C jmpbuf.
  - <catch-method> subtype of <method>
    - C func invokes longjmp().

Continuations

- Save/restore C stack segments since last continuation (see ccont library).
- Flush <message> and value stack buffers.
- Save/restore <message> and value stack pointers.
- Save/restore fluid variable bindings.
- Save/restore current catch.

Interpreter Evaluation

- All (eval) expressions are byte-compiled
  - Phase 1: macro-expansion
    - Simple quasiquote expander.
  - Phase 2: intermediate representation
    - (lambda ...) => (%add-method <object> ... (%method ...))
    - (%method slots formals ...) => <%ir ...>
  - Phase 3: semantics
    - car-position lambda inlining
    - closure generation and environment planning
    - constant-folding
  - Phase 4: assembly generation

<method>

- Abstract base class.
- Slots:
  - func: points to address of primitive C function.
  - formals: parameter list (for debugging)
  - code: code value (for debugging)
  - properties: association list. eg.:

    ```
    (
        (documentation . "...")
        (no-side-effect? . #t)    ☼
    )
    ```

<byte-code-method>

- Supers: <method>
- Slots:
  - func: Primitive method function points to primitive byte-code execution function.
  - code: A byte-code <string>
  - constants: A constants <vector>
    - Contents be shared between multiple methods defined in an expression:
      - A <vector> of global symbols.
      - A <vector> of global <bindings> for each global symbol.
      - Other (quote)'ed expressions or constants.

Constant Folding

- (constant-fold symbol scope) =>
  - If symbol is a <symbol> AND
  - If symbol is bound to a global AND
  - If (readonly? binding) THEN
    - (value binding) ELSE op
- (constant-fold (op . args) scope) =>
  - If (constant-fold op scope) is constant? AND
  - If all (map constant-fold args) are constant? AND
  - If (%lookup op rcvr) has no-side-effect? THEN
    - (eval-no-constant-fold (cons op args)) ELSE
    - (cons op args)

Without Constant Folding

- (define x (cons 1 2))
  (lambda (y . args) (set! (car x) y))
- Bytecode listing with its value stack:

```
(    (probe 0)              ;       value stack
 (nargs-rest- 1)            ; temps          | args
     (probe 3)              ;                | y
 (arg 0 y)                  ; &y             | y
 (contents)                 ; y              | y
 (glo 2 x)                  ; x y            | y
     (probe 2)              ; x y            | y
 (glo 1 car)                ; car x y        | y
 (glo 0 setter)             ; setter x y     | y
 (call 1)                   ; set-car! x y   | y
 (call-tail 2)              ;                |
 (rtn))
```

With Constant Folding

- (define x (cons 1 2))
  (lambda (y . args) (set! (car x) y))
-
```
(    (probe 0)                          ;       value stack
 (nargs-rest- 1)                        ; temps           | args
     (probe 3)                          ;                 | y
 (arg 0 y)                              ; &y              | y
 (contents)                             ; y               | y
 (glo 2 x)                              ; x y             | y
 (const 4 #<operation set-car!>)        ; set-car! x y    | y
 (call-tail 2)                          ;                 |
 (rtn))
```

Tak Benchmark

- Recursion, Fixnum +, -
  - ```
    (define (tak x y z)
      (if (not (< y x))
          z
          (tak (tak (- x 1) y z)
               (tak (- y 1) z x)
               (tak (- z 1) x y))))
    ```
    ```

Performance Comparision

- (tak 18 12 6) (tak 30 15 9)
  (tak 33 15 9) (tak 40 15 9)
  - ikarus 0.25 sec
  - chicken 1.41 sec (precompiled)
  - oaklisp 2.63 sec
  - mzscheme 2.65 sec
  - scheme-r5rs 5.89 sec
  - guile 7.91 sec
  - larceny 10.35 sec
  - LL 12.93 sec

Future Work

- Vary more levels of self-reference to improve performance?
- Inline primitives into byte-code executor?
- Imbed TinyCC C compiler?
- Re-implement using COLA libid?
- Scheme is the New Assembler?
- Re-implement LL as a library of Scheme procedures and syntax using Ikarus?

Conclusion

- Q & A
- Discussion

Resources

- ☼
  - http://github.com/kstephens/ll/tree/master
  - http://kurtstephens.com/page/article.html/5
  - http://kurtstephens.com/node/53
  - http://www2.lib.uchicago.edu/~keith/crisis/benchmarks/tak/
  - http://www.cs.indiana.edu/~aghuloum/ikarus/
  - The Art of The Metaobject Protocol - Gregor Kiczales
  - http://www.amazon.com/Brain-Makers-HP-Newquist/dp/0672304120