Implementing Functional Languages

e.g. Lisp, Scheme, ML, Haskell, Miranda

Uniform, polymorphic references to data
- dynamic typing, as in Lisp & Scheme (and Smalltalk & ...)
- variables of polymorphic type, as in ML & Haskell

⇒ uniform “boxed” representation of all data objects,
tagged pointers to encode some types (e.g. ints) cheaper

First-class, lexically-nested functions
- static scoping of nested functions
  ⇒ closures to represent function values
- functions can outlive defining scope
  ⇒ heap-allocated environments
- calls of computed expressions
  ⇒ (fancier) call graph analysis

Heavy use of recursion instead of iteration
⇒ tail call, tail recursion elimination

Immutable update-by-copy data structures
⇒ version arrays, compile-time reference counting

Miranda & Haskell: lazy evaluation
⇒ strictness analysis

Run-time typing

In many higher-level languages, need to
- ... treat all/many values uniformly
  - dynamically typed code
  - (parametric or subtype-)polymorphic code
- ... be able to determine the type of the value at run-time
  - resolve dynamically dispatched messages, subtype tests
  - perform run-time type checks
  - support precise GC, reflection, ...

An approach: boxing + type field
- represent all values as one-word pointers to data structures
- add implicit type field to each object
  - encoded as small enumerated tag, class pointer, virtual function table, ...

\+ all code can handle any data

\+ can always determine type at run-time

– space cost for type field

– very slow if have to box scalars like ints, floats, chars, bools

Tagging

Observation: not all bits of pointer are used
- alignment often requires 2-3 low-order bits to be 0
- high-order bits often all the same,
  since full address space isn’t needed

Idea: use those bits to encode type tag for most common types
- strip out type info before dereferencing pointer

\+ saves a word of space in the target object
  - good for small objects, like ints, floats, cons cells, pairs, ...

\+ speeds type-testing code

– slows pointer dereference time to extract real pointer from tagged pointer

Naive asm code, assuming low-order tag for pairs:

```
%ptr = %tagged_ptr - pair_tag;
%first = *(%ptr + 0);
%second = *(%ptr + 4);
```

Cooler code: combine untagging with field offset calculation

```
%first = *(%tagged_ptr + (0 - pair_tag));
%second = *(%tagged_ptr + (4 - pair_tag));
```

Untagging is free!

Tagged scalars

Further idea: for one-word immutable values (ints, chars, bools), store the value in the pointer word itself!

E.g. 2-3 low-order bits for type tag, 29-30 high-order bits for value
- left-shift real value by 2-3, then add in tag, to tag a value
- subtract tag, then right-shift by 2-3, to untag

\+ no memory dereferencing to get value

\+ no memory allocation cost when doing arithmetic

– some cost to manipulate tags

– lose 2-3 bits of precision
  - find for chars, bools
  - OK for ints (except when manipulating memory words)
  - bad for floats (e.g. rounding is hard to get right)

Cool trick: choose all-zero as the tag for ints

Then:
- tagged ints can be added, subtracted, & compared directly, w/o untagging first!
- tagged ints can be multiplied & divided by adding one shift
- overflow behavior preserved

Implementing first-class lexically nestable functions

Functions are first-class data values
- can be passed as arguments, returned from fns, stored in data structures
- potentially anonymous
- lexically-scoped

Example:

```
(define mul-by (lambda lst n)
  (map (lambda (x) (* x n)) lst))
```

2 components of a function value (a **closure**):
- code pointer
- lexically-enclosing environment pointer

Steps in deciding how to implement a closure:
- **strategy analysis**: where to allocate closure
- **representation analysis**: how to lay out data structure

Strategy analysis

Option 1: heap allocation
+ most general option
+ simple decision to make
– expensive to create, invoke, and reclaim closure
– may require heap-allocation of lexically-enclosing env

Supports “upward funargs”

Example:

```
(define (add x) (lambda (y) (+ x y)))
(define inc (add 1))
(define dec (add -1))
(print (inc (dec 3)))
```

Stack allocation

Option 2: stack allocation

If closure’s dynamic extent is contained within the extent of its lexically-enclosing activation record, then can allocate closure as part of a.r.’s stack frame (a LIFO closure)

+ faster allocation, free reclamation
+ enclosing environment can be stack-allocated

– invocation still slow

Inlining calls to closures

Option 3: represent closure in-line

If invoking a known closure, inline-expand body

If all uses of a closure inlined away, don’t create closure
- closure’s environment turns into local variables

+ free allocation, fast invocation, free reclamation

Enables closure-based user-defined control structures

Escape analysis

Determine if closure (or any data structure)
has LIFO extent, i.e. does not **escape** stack frame

- use stack allocation for non-escaping data structures

Track flow of value, see where it goes

Has LIFO extent (i.e., doesn't escape):

- when created
- when assigned to local variable
- when invoked

A hard case:

- passed as argument to function
  - if intraprocedural analysis: conservatively assume escapes
  - if interprocedural analysis: may or may not escape

Harder cases:

- returned
- stored in global/non-local variable or (escaping) data structure

Assume escapes

Interprocedural escape analysis

Compute for each formal parameter whether that parameter **escapes**

Construct program's call graph

Initialize all formals to "does not escape"

Initialize worklist to empty set

Process each function:
if formal parameter labeled "does not escape"
escapes locally within this function,
change formal to "escapes" and put all callers on worklist

While worklist non-empty:
remove function from worklist, reprocess

- at call site, actual argument escapes if corresponding formal escapes

---

Representation analysis

How to represent closure's lexical environment?

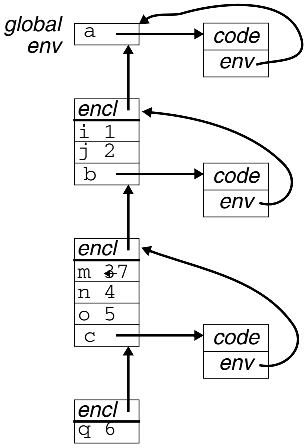
Option 1: **deep binding**

- store pointer to enclosing environment
- share enclosing environment across all nested environments & closures

---

Example of deep binding

```
(define a (lambda (i j)
  (define b (lambda m n o)
    (define c (lambda q)
      (+ q m i)) ;; here
    (set! m 7)
    (c 6))
  (b 3 4 5))
(a 1 2)
```



Representation analysis, cont

Option 2: **shallow binding**

- copy needed values into environment when created
- \+ faster access to lexically enclosing vars
- – bigger environments
- – slower to create environments

Option 3: **very shallow binding**

- copy needed values into closure itself

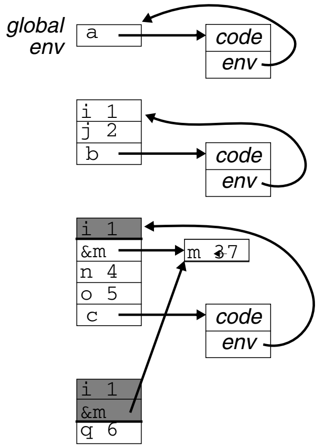
Cannot copy values of mutable variables
⇒ do **assignment conversion** first

- replace mutable variable with pointer to heap-allocated reference cell
- \+ can copy the pointer freely
- – space overhead
- – extra indirection

⇒ best for mostly functional code, e.g. Scheme

Example of shallow binding

```
(define a (lambda (i j)
  (define b (lambda m n o)
    (define c (lambda q)
      (+ q m i)) ;; here
    (set! m 7)
    (c 6))
  (b 3 4 5))
(a 1 2)
```



Restricted semantics

If only allow to pass nested fns down, but not return them, then closures & environments are LIFO

- environment can be stack-allocated, not heap-allocated
- e.g. Pascal, Modula-3
  (and Vortex's broken default for Cecil)

If allow nested procedures but not first-class procedures, then don't need closure data structures

- do not need pair, just extra implicit environment argument
- e.g. Ada

If allow first-class procedures but no nesting, then no lexically enclosing environment needed

- implement function value with just a code address
- e.g. C, C++