Values and Locations

Conventional imperative languages manipulate the state of the machine, so it is not surprising that we need to consider locations of data in memory.

Variables are normally bound to locations, which contain values.

- A large value, like a record or array, might require several locations.

Values themselves are sometimes represented by locations, depending on the type of value and the language.

- Some values are implicitly represented by locations (e.g., the pair values from the last lecture).

- Some languages allow us to manipulate locations explicitly (e.g., pointers in C/C++.)
Handling Large Values

Real machines are very efficient at handling small, fixed-size chunks of data, especially those that fit in a single machine word (e.g. 16-64 bits), which usually includes:

- Numbers, characters, booleans, enumeration values, etc.
- Memory addresses (locations).

But often we want to manipulate larger pieces of data, such as records and arrays, which may occupy many words.

There are two basic approaches to representing larger values:

- The *direct* representation uses as many words as necessary to hold the contents of the value.
- The *indirect* representation of a large value *implicitly* uses a pointer to the (first of the) locations holding the contents.

For example, an array of 100 integers could be directly represented by 100 words holding the contents of the array, or indirectly represented by an implicit 1-word pointer to 100 consecutive locations holding the array contents.

The language’s choice of representation makes a big difference to the semantics of operations on the data, e.g.:

- What does assignment mean?
- How does parameter passing work?
- What do equality comparisons mean?
Direct Representation Semantics

Earlier languages often used direct representation semantics for records and arrays. For example, in Pascal and related languages,

```pascal
TYPE Employee =
RECORD
  name : ARRAY (1..80) OF CHAR;
  age : INTEGER;
END;
```

specifies a direct representation, in which value of type EMPLOYEE will occupy 84 bytes (assuming 1 byte characters, 4 byte integers).

The semantics of assignment is to copy the entire representation. Hence the code

```pascal
VAR e1,e2 : Employee;
...
e1.age := 91;
e2 := e1;
e1.age := 19;
WRITE(e1.age, e2.age);
```

prints 19 followed by 91.
Direct Representation Problems

Assignment using the direct representation has appealing semantics, but two significant problems:

- Assignment of a large value is expensive, since lots of words may need to be copied.

- Since compilers need to generate code to move values, and (often) allocate space to hold values temporarily, they need to know the size of the value.

These problems make the direct representation unsuitable for value of arbitrary size. For example, direct representation works fine for pairs of integers, but not for pairs of arbitrary values that might themselves be pairs.
Indirect Representation Semantics

ML implementations \textbf{implicitly} allocate tuples and \texttt{datatype} values on the heap, and represent record \texttt{values} by \texttt{references} (pointers) into the heap. Java does the same thing with objects (although we must say \texttt{new} explicitly at points of allocation).

As a natural result, both languages use \texttt{shallow copy} semantics for assignment and argument passing. Example:

```java
class emp {
    String name;
    int age;
}
emp e1;
e1.age = 91;
emp e2 = e1;
e1.age = 18;
System.out.print(e2.age);
```

prints 18

If you want to copy the entire contents of record or class, you must do it yourself, element by element (though Java objects do have a standard library method called \texttt{clone} to do the job).

Neither language allows user programs to manipulate the internal pointers directly. And neither supports explicit \texttt{deallocation} of records (or objects) either; both provide automatic \texttt{garbage collection} of unreachable heap values.
Explicit Pointers

Many languages that use direct semantics also have separate **pointer types** to enable programmers to construct recursive data structures, e.g., in C:

```c
typedef struct intcell *intlist;
struct intcell {
    int head;
    intlist tail;
}
intlist mylist =
    (intlist) malloc(sizeof(struct intcell));
while (list != NULL)
    if (list->head != i) then
        list = list->tail;
```

In most such languages, pointers are restricted to addresses returned by allocation operations, but C/C++ allows the address of **anything** to be taken and later dereferenced, and supports **pointer arithmetic**. While this feature can support very efficient code, it also destroys the safety of the type system.
Lifetime vs. Scope

Typically, a computation requires more locations over the course of its execution than the target machine can efficiently provide — but at any given point in the computation, only some of these locations are needed.

Thus nearly all language implementations support the \textbf{re-use} of locations that are no longer needed.

The \textbf{lifetime} of an allocated piece of memory (loosely, an “object”) runs from the time when it is allocated into one or more locations to the time when the location(s) get re-used.

For the program to work, the lifetime of an object should last as long as it is “in use.” More precisely:

\begin{itemize}
  \item An object representing a variable should live as long as the variable is still in scope.
  \begin{itemize}
    \item This is normally enforced by the language implementation.
    \item E.g., a function’s local variables are usually allocated in a stack frame whose lifetime lasts from the time the function is called to the time it returns — exactly corresponding to the variable’s scope.
  \end{itemize}

  \item An object whose location is itself a value (implicit or explicit) should live as long as the value is accessible. (Normally, values are accessible from variables or fields in other values.)
  \begin{itemize}
    \item This is trickier to enforce, unless the language uses a garbage collector.
  \end{itemize}
\end{itemize}
Problems with Lifetimes

If the language supports pointers and explicit deallocation is allowed, it is easy for the programmer to accidentally kill off an object even though it is still accessible, e.g.:

```c
char *foo() {
    char *p = malloc(100);
    free(p);
    return p;
}
```

Here the allocated storage remains accessible (via the value of variable `p`) even after that storage has been freed (and possibly reallocated for something else).

This is usually a **bug** (a **dangling pointer**). The converse problem, failing to deallocate an object that is no longer needed, can cause a **space leak**, leading to unnecessary failure of a program by running out of memory. Using a **garbage collector** avoids both problems.

Languages like C/C++ that permit the address of anything to be used as a pointer can cause even worse problems:

```c
int *foo() {
    int x = 1;
    return &x;
}
```

Here `foo` returns a pointer to its own local variable `x`, but the storage for `x` will almost certainly be overwritten by the next function call.
Storage Classes for Data

Runtime storage can be classified based on the desired lifetime of the data being stored.

Static Data: Permanent Lifetimes

- Global variables and constants.
- Allows fixed address to be compiled into code.
- No runtime management costs.
- Original FORTRAN (no recursion) used static activation records.

Stack Data: Nested Lifetimes

- Allocation/deallocation is cheap (just adjust stack pointer).
- Most architectures support cheap sp-based addressing.
- Good locality for VM systems, caches.
- C, Algol/Pascal family, Java use stack for activation records.

Heap Data: Arbitrary Lifetimes

- Requires explicit allocation and (dangerous) explicit deallocation or garbage collection.
- Lisp, ML, many interpreted languages need heap for activation records, which have non-nested lifetimes.
Procedures and Functions

- Subroutines: avoid duplicating frequently used code
- Procedural abstraction: divide programs into named components with hidden internals
- Provide framework for managing local data, especially in recursive programs

Invoking a procedure generates a **procedure activation**, which has associated data such as:

- the **return address** of the caller
- the **actual** values corresponding to the **formal** parameters of the procedure
- space for the values of **local variables** associated with the procedure.

The data associated with each activation is independent from that for all other activations, so many activations can exist at once, as happens in a recursive program.

In most languages, activation data can be stored on a **stack**, and we speak of pushing and popping activation **frames** from the stack, which is a very efficient way of managing local data.
Activation Frames

A typical activation stack, shown just before inner call to \( f \) returns.

Note that the offsets of parameters and local variables from the frame pointer (\( fp \)) are statically known, allowing very efficient access at runtime.
What about Registers?

Although it is convenient to view all locations as memory addresses, most machines also have registers, which are:

- much faster to access,
- but very limited in number (e.g., 4 to 64).

So compilers try to keep variables (and pass parameters) in registers when possible, but always need memory as a backup. Using registers is fundamentally just an (important!) optimization.

Easy to have environment map each name to location that is either memory address or register.

- But registers don’t have addresses, so they can’t be accessed indirectly, and register locations can’t be passed around or stored.
Tail-call optimization

Any iteration can be written as a recursion.

For example:

\[
\text{while } (t) \text{ do } e
\]

is equivalent to

\[
\text{void } f (\text{bool } b) \{ \\
\quad \text{if } (b) \text{ then } \{ \\
\quad\quad e; \\
\quad\quad f(t) \\
\quad \} \\
\}
\]

\[
f(t)
\]

where we assume that the variables used by \(e\) and \(t\) are global.

When can we do the converse? It turns out that a recursion can be rewritten as an iteration whenever all the recursive calls are in tail position. To be in tail position, the call must be the last thing performed by the caller before it itself returns.
Tail-call Examples

List operations can often be made tail-recursive in this way:

(* tail-recursive *)
fun last [x] = x
    | last (x::xs) = last xs

(* not tail-recursive *)
fun length [] = 0
    | length (x::xs) = 1 + (length xs)

(* use accumulating parameter; now is tail-recursive *)
fun length l =
    let fun f ([],len) = len
        | f (x::xs,len) = f (xs,len+1)
    in f (l,0)
end

A decent compiler can turn tail-calls into iterations, thus saving the cost of pushing an activation frame on the stack. This is essential for languages (like ML) that lack iteration, and useful even for those that have it (like C).
Systematic Removal of Recursion

(Adapted from Sedgewick, *Algorithms*, 2nd ed.. Examples in C.)

One way to get a better appreciation for how recursion is implemented is to see what is required to get rid of it.

Original program:

```c
typedef struct tree *Tree;
struct tree {
    int value;
    Tree left;
    Tree right;
};

void printtree(Tree t) {
    if (t) {
        printf("%d\n", t->value);
        printtree(t->left);
        printtree(t->right);
    }
}
```
Step 1:

Remove **tail-recursion**.

```c
void printtree(Tree t) {
    // Step 1: Remove tail-recursion.
    top:
        if (t) {
            printf("%d\n", t->value);
            printtree(t->left);
            t = t->right;
            goto top;
        }
}
```

Step 2:

Use explicit stack to replace non-tail recursion. Simulate behavior of compiler by pushing local variables and return address onto the stack **before** call and popping them back off the stack **after** call.

Here there is only one local variable (t) and the return address is always the same, so there’s no need to save it.
Stack empty;
void push(Stack s, Tree t);
Tree pop(Stack s);
int isEmpty(Stack s);

void printtree(Tree t) {
    Stack s = empty;
    top:
        if (t) {
            printf("%d\n", t->value);
            push(s, t);
            t = t->left;
            goto top;
        } else {
            if (!isEmpty(s)) {
                t = pop(s);
                goto retaddr;
            }
        }
    retaddr:
        t = t->right;
        goto top;
}
}
Step 3:

Simplify by:

- Rearranging to avoid the retaddr label.
- Pushing right child instead of parent on stack.
- Replacing first goto with a while loop.

```c
void printtree(Tree t) {
    Stack s = empty;
    top:
    while (t) {
        printf("%d\n", t->value);
        push(s, t->right);
        t = t->left;
    }
    if (!(isEmpty(s))) {
        t = pop(s);
        goto top;
    }
}
```
Step 4:

Rearrange some more to replace outer goto with another while loop.

(This is slightly wasteful, since an extra push, stackempty check and pop are performed on root node.)

```c
void printtree(Tree t) {
    Stack s = empty;
    push(s,t);
    while(!(isEmpty(s))) {
        t = pop(s);
        while (t) {
            printf("%d\n",t->value);
            push(s,t->right);
            t = t->left;
        }
    }
}
```
Step 5:

A more symmetric version can be obtained by pushing and popping the left children too.

Compare this to the original recursive program.

```c
void printtree(Tree t) {
    Stack s = empty;
    push(s,t);
    while(!(isEmpty(s))) {
        t = pop(s);
        if (t) {
            printf("%d\n",t->value);
            push(s,t->right);
            push(s,t->left);
        }
    }
}
```
Step 6:

We can also test for empty subtrees before we push them on the stack rather than after popping them.

```c
void printtree(Tree t) {
    Stack s = empty;
    if (t) {
        push(s, t);
        while(!(isEmpty(s))) {
            t = pop(s);
            printf("%d
", t->value);
            if (t->right)
                push(s, t->right);
            if (t->left)
                push(s, t->left);
        }
    }
}
```


Procedure Parameter Passing

When we activate a procedure, the formal parameters get bound to locations containing values.

• How is this done and which locations are used?
• Do we pass addresses or contents of variables from the caller?
• How do we pass actual values that aren’t variables?
• What does it mean to pass a large value like an array?

Two main approaches:

• call-by-value
• call-by-reference
Call-by-Value

• Each actual argument is evaluated to a value before call.

• On entry, value is bound to formal parameter just like a local variable.

• Updating formal parameter doesn’t affect actuals in calling procedure.

```c
double hyp(double a, double b) {
    a = a * a;
    b = b * b;
    return sqrt(a+b);
}
```

• Simple; easy to understand!

• Implement by binding the formal parameters to freshly-allocated locations, and and copying the actual values into these locations (just like assignment).
Problems with Call-by-Value

- Can be inefficient for large directly-represented values:

Example (C): Calls to \texttt{dotp} copy 20 doubles

```c
typedef struct {double a1,a2,...,a10;} vector;
double dotp(vector v, vector w) {
    return v.a1 * w.a1 + v.a2 * w.a2 + ... + v.a10 * w.a10;
}
vector v1,v2;
double d = dotp(v1,v2);
```

- Cannot affect calling environment directly. (Of course, perhaps this is a \textbf{good} thing!)

Example: calls to \texttt{swap} have no effect:

```c
void swap(int i,int j) {
    int t;
    t = i ; i = j; j = t;
}
...
swap(a[p],a[q]);
```

- Can at best \textbf{return} only one result (as a value), though this might be a record.
Call-by-Reference

- Pass the existing location of each actual parameter.
- On entry, the formal parameter is bound to this location, which must be dereferenced to get value, but can also be updated.
- If actual argument doesn’t have a location (e.g., “2 + 3”), either:
  - Evaluate it into a temporary location and pass address of temporary, or
  - Treat as an error.
- Now swap, etc., work fine!
- Accesses are slower.
- Lots of opportunity for aliasing problems, e.g.,

  \[
  \begin{align*}
  \text{PROCEDURE} \ & \ matmult(a,b,c: \text{ MATRIX}) \\
  \ & \ \ldots \ (* \ \text{sets} \ c := a \ * \ b *) \\
  \ & \ matmult(a,b,a) \ (* \ \text{oops!} \ *)
  \end{align*}
  \]

- Call-by-value-result (a.k.a. copy-restore) addresses this problem, but has other drawbacks.
Hybrid Methods; Records and Arrays

How might we combine the simplicity of call-by-value with the efficiency of call-by-reference, especially for large directly-represented values?

- In Pascal, Ada, and similar languages, where records and arrays are both represented directly, the programmer can specify (in the procedure header) for each parameter whether to use call-by-value or call-by-reference.

- In ANSI C/C++, record (struct or class) values are represented directly, but arrays are represented indirectly. C always uses call-by-value, but programmers can take the address of a variable explicitly, and pass that to obtain cbr-like behavior:

```c
swap(int *a, int *b) {
    int t;
    t = *a; *a = *b; *b = t;
}
swap (&a[p], &a[q]);
```

Of course, it is the programmer’s responsibility to make sure that the address remains valid (especially when it is returned from a function).
Complex and Simple Solutions

- C++ supports both cbr parameters and explicit pointers:

```c
swap(int &a, int *b) {
    int t;
    t = a; a = *b; *b = t;
}
...
swap(a[p], &a[q]);
```

Mixing explicit and implicit pointers can be very confusing!

- In Java and ML, values of both records (objects) and arrays are represented indirectly. These languages have only call-by-value, but this doesn’t actually cause copying, even for record or array values.

- Approach is made more feasible because programmer doesn’t have to worry about lifetime of heap data, due to automatic garbage collection.

- Clever compilers can decide whether smallish objects should be heap-allocated or manipulated directly, while continuing to give the semantic effect of indirect representation.
Substitution

- Can often get the effect we want using substitution, i.e., macro-expansion, e.g. (in C):

```c
#define swap(x,y) {int t; t = x; x = y; y = t;}
...
swap(a[p], a[q]);
```

- **BUT** blind substitution is dangerous because of possible "variable capture," e.g.,

```c
swap(a[t], a[q])
```

expands to

```c
{int t; t = a[t]; a[t] = a[q]; a[q] = t;}
```

Here "t is captured" by the declaration in the macro, and is undefined at its first use.

- Really want "substitution with renaming where necessary" = **call-by-name** (first proposed in Algol60).

- Flexible, but potentially very confusing, and inefficient to implement.

- If language has no mutable variables (as in "pure" functional languages), substitution gives a beautifully simple semantics for procedure calls.
Exceptions

Programs often need to handle exceptional conditions, i.e., deviations from “normal” control flow.

Exceptions may arise from

- failure of built-in or library operations (e.g., division by zero, end of file)
- user-defined events (e.g., key not found in dictionary)

Awkward or impossible to deal with these conditions explicitly without distorting normal code.

Most recent languages (Ada, C++, Java, etc.) provide a means to define, raise, and handle exceptions.

Java example:

```java
class Help extends Exception {}
try {
    ...
    if (gone wrong) throw new Help();
    ...
    x = a / b; ...
} catch (Help e) {
    ...
    report problem...
} catch (ArithmeticException e) {
    x = -99;
}
```
What to do in an exception?

If there is a statically enclosing handler, the thrown exception behaves much like a `goto`. In previous example:

```java
... 
if (gone wrong) goto help_label; 
.. 
help_label: ...report problem...
```

But what if there is no handler explicitly wrapped around the exception-throwing point?

- In most languages, uncaught exceptions **propagate** to next dynamically enclosing handler. E.g, caller can handle uncaught exceptions raised in callee.

- Many languages permit a value to be returned along with the exception itself.

- A few languages support **resumption** of the program at the point where the exception was raised.
Exception Handling Example

class BadThing extends Exception {};

int foo () {
    ... throw new BadThing(); ... 
}

bar () {
    int x;
    try {
        x = foo ();
    } catch (BadThing e) {
        x = 0;
    }
}
Exceptions vs. Error Values

An alternative to user-raised exceptions is to return status values, which must be checked on return:

```haskell
fun find (k0:string) (env: (string * int) list) : int option =
  case env of
  nil => NONE
  | (k,v)::t =>
    if k = k0 then
      SOME v
    else find k0 t

... case find "abc" e0 of
  SOME v => ... v ...
  | NONE => ... error code ...
```
Exception vs. Error Values (2)

With exceptions, we can defer checking for (rare) error conditions to a more convenient point.

```
exception NotFound
fun find (k0:string)
  (env: (string * int) list)
  : int =
  case env of
    nil => raise NotFound
  | (k,v)::t =>
    if k = k0 then
      v
    else find k0 t

  ... let val v = find "abc" e0
  in ... v ...
  end
  handle NotFound => ...error code...
```
Exceptions and the Stack

- To implement exceptions, runtime system can maintain a handler stack with an entry for each handler context currently alive. When an exception occurs, control passes to the top handler on the stack. If this handler itself raises an exception, control passes to the next handler on the stack, etc.

- In this execution model, raising an exception provides a way to return quickly from a deep recursion, with no need to pop stack frames one at a time.

Example:

```haskell
fun product l = 
  let exception Zero 
    fun prod l = 
      case l of 
        nil => 1 
      | (h::t) => 
        if h = 0 then raise Zero 
        else h * (prod t) 
      in (prod l) handle Zero => 0 
  end 
```

- Alternatively, some runtime systems use a static table from code addresses to handler addresses. If an exception occurs (assumed to be rare), the table is inspected to find the appropriate handler. If there is no handler defined in the current routine, the runtime system looks for a handler that covers the return address (in the caller), and so on up the call-stack.