CS558 Programming Languages
Winter 2006
Lecture 3b
**Environments and Stores, Formally**

- We write $E(x)$ means the result of looking up $x$ in environment $E$. (This notation is because an environment is like a function taking a name as argument and returning a meaning as result.)

- We write $E + \{x \mapsto v\}$ for the environment obtained from existing environment $E$ by extending it with a new binding from $x$ to $v$. If $E$ already has a binding for $x$, this new binding replaces it.

The **domain** of an environment, $\text{dom}(E)$, is the set of names bound in $E$. Analogously with environments, we’ll write

- $S(l)$ to mean the value at location $l$ of store $S$

- $S + \{l \mapsto v\}$ to mean the store obtained from store $S$ by extending (or updating) it so that location $l$ maps to value $v$.

- $\text{dom}(S)$ for the set of locations bound in store $S$.

Also, we’ll write

- $S - \{l\}$ to mean the store obtained from store $S$ by removing the binding for location $l$. 
So far, we’ve presented operational semantics using interpreters. These have the advantage of being precise and executable. But they are not ideally compact or abstract.

Another way to present operational semantics is using state transition judgments, for appropriately defined machine states.

For example, consider a simple language of imperative expressions, in which variables must be defined before use, using a local construct.

```
exp := var | int
    | ’( ’ ’+’ exp exp ’)’
    | ’( ’ ’local’ var exp exp ’)’
    | ’( ’ ’:=’ var exp ’)’
    | ’( ’ ’if’ exp exp exp ’)’
    | ’( ’ ’while’ exp exp ’)’
    | etc.
```

Informally, the meaning of (local x e₁ e₂) is: evaluate e₁ to a value v₁, create a new store location l bound to x and initialized to v₁, and evaluate e₂ in the resulting environment and store.
To evaluate this language, we choose a machine state consisting of:

- the current **environment** \( E \), which maps each in-scope variable to a location \( l \).
- the current **store** \( S \), which maps each location \( l \) to an integer value \( v \).
- the current **expression** \( e \), to be evaluated.

We give the state transitions in the form of judgments:

\[
\langle e, E, S \rangle \Downarrow \langle v, S' \rangle
\]

Intuitively, this says that evaluating expression \( e \) in environment \( E \) and store \( S \) yields the value \( v \) and the (possibly) changed store \( S' \).
To describe the machine’s operation, we give **rules of inference** that state when a judgment can be derived from judgments about sub-expressions.

The form of a rule is

\[
\text{premises} \quad \frac{}{\text{conclusion}} \quad \text{(Name of rule)}
\]

We can view evaluation of the program as the process of building an inference tree.

This notation has similarities to axiomatic semantics: the notion of derivation is essentially the same, but the content of judgments is different.
EVALUATION RULES (1)

\[
\begin{align*}
  l &= E(x) \quad v = S(l) \quad \text{(Var)} \\
  \langle x, E, S \rangle &\downarrow \langle v, S \rangle \\
  \langle i, E, S \rangle &\downarrow \langle i, S \rangle \quad \text{(Int)} \\
  \langle e_1, E, S \rangle &\downarrow \langle v_1, S' \rangle \quad \langle e_2, E, S' \rangle &\downarrow \langle v_2, S'' \rangle \quad \text{(Add)} \\
  \langle (+ e_1 e_2), E, S' \rangle &\downarrow \langle v_1 + v_2, S'' \rangle \\
  \langle e_1, E, S \rangle &\downarrow \langle v_1, S' \rangle \quad l \notin \text{dom}(S') \quad \langle e_2, E + \{x \mapsto l\}, S' + \{l \mapsto v_1\} \rangle &\downarrow \langle v_2, S'' \rangle \quad \text{(Local)} \\
  \langle \text{(local} x \mathit{e_1 \ e_2}), E, S \rangle &\downarrow \langle v_2, S'' - \{l\} \rangle \\
  \langle e, E, S \rangle &\downarrow \langle v, S' \rangle \quad l = E(x) \quad \text{(Assgn)} \\
  \langle (:= x \ e), E, S \rangle &\downarrow \langle v, S' + \{l \mapsto v\} \rangle
\end{align*}
\]
\[
\langle e_1, E, S \rangle \downarrow \langle v_1, S' \rangle \quad v_1 \neq 0 \quad \langle e_2, E, S' \rangle \downarrow \langle v_2, S'' \rangle \quad (\text{If-nzero})
\]

\[
\langle (\text{if } e_1 \ e_2 \ e_3), E, S \rangle \downarrow \langle v_2, S'' \rangle
\]

\[
\langle e_1, E, S \rangle \downarrow \langle 0, S' \rangle \quad \langle e_3, E, S' \rangle \downarrow \langle v_3, S''' \rangle \quad (\text{If-zero})
\]

\[
\langle (\text{if } e_1 \ e_2 \ e_3), E, S \rangle \downarrow \langle v_3, S''' \rangle
\]

\[
\langle e_1, E, S \rangle \downarrow \langle v_1, S' \rangle \quad v_1 \neq 0 \quad \langle e_2, E, S' \rangle \downarrow \langle v_2, S'' \rangle \quad (\text{While-nzero})
\]

\[
\langle (\text{while } e_1 \ e_2), E, S'' \rangle \downarrow \langle v_3, S''' \rangle
\]

\[
\langle e_1, E, S \rangle \downarrow \langle 0, S' \rangle \quad (\text{While-zero})
\]

\[
\langle (\text{while } e_1 \ e_2), E, S \rangle \downarrow \langle 0, S'' \rangle
\]
Notes on the Rules

- The structure of the rules guarantees that at most one rule is applicable at any point.
- The store relationships constrain the order of evaluation.
- If no rules are applicable, the evaluation gets stuck; this corresponds to a runtime error in an interpreter.

We can view the interpreter for the language as implementing a bottom-up exploration of the inference tree. A function like

```plaintext
Value eval(Exp e, Env env) { .... }
```

returns a value \( v \) and has side effects on a global store such that

\[
\langle e, \text{env}, \text{store}_{\text{before}} \rangle \Downarrow \langle v, \text{store}_{\text{after}} \rangle
\]

The implementation of `eval` dispatches on the syntactic form of \( e \), chooses the appropriate rule, and makes recursive calls on `eval` corresponding to the premises of that rule.

Question: how deep can the derivation tree get?
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.
**DIRECT VS. INDIRECT**

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?
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

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

prints 19 followed by 91.
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.
ML implementations implicitly allocate tuples and datatype values on the heap, and represent record values by references (pointers) into the heap. Java does the same thing with objects (although we must say new explicitly at points of allocation).

As a natural result, both languages use 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;
e1.age = 18;
System.out.print(e2.age);

prints 18
```
If you want to copy the entire contents of record or object, you must do it yourself, element by element (though Java objects do have a standard library method called `clone` to do the job).

Neither language allows user programs to manipulate the internal pointers directly. And neither supports explicit `deallocation` of records (or objects) either; both provide automatic `garbage collection` of unreachable heap values.
Many languages that use direct semantics also have separate **pointer types** to enable programmers to construct recursive data structures, e.g.:

```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 language safety.
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 **re-use** of locations that are no longer needed.

The **lifetime** of an allocated piece of memory (loosely, an “object”) extends 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 the object is (potentially) being used.
More precisely:

- An object whose location is bound to a variable should live as long as the variable is still in scope.
  
  – This is normally enforced by the language implementation.
  
  – E.g., a function’s local variables are typically bound to locations 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.

- 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.)
  
  – This is trickier to enforce, unless the language uses a garbage collector.
PROBLEMS WITH EXPLICIT CONTROL OF 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.
PROBLEMS WITH UNRESTRICTED POINTERS

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 Based on Data Lifetime

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
- Needs explicit allocation and (dangerous) explicit deallocation or G.C.
- Lisp, ML, many interpreted languages need heap for activation records, which have non-nested lifetimes.