CS302 Spr'99 Lecture Notes
Lecture 9

Procedure Parameter Passing

```
TYPE IARRAY IS ARRAY OF INTEGER;
PROCEDURE f(x:INTEGER,y:INTEGER) IS ...
PROCEDURE g(z:IARRAY,q:IARRAY) IS ...
VAR a: IARRAY, w: INTEGER
.
.
.
f(3,w); ... g(a,a); ... f(17+5,a[3]);
.
.
.
```

• Do we pass addresses (**l-values**) or contents (**r-values**) of variables?

• How do we pass actual values that aren't variables?

• What does it mean to pass an aggregate value like an array?

Call-by-Value (i.e., r-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.

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

• Simple; easy to understand!

• Implement by copying.

Problems with Call-by-Value

• Can be inefficient if value is large.

Example: Calls to `dotp` copy 20 doubles:

```
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. (Often a **good** thing, but not always!)

Example: calls to `swap` have no effect:

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

• Can at best **return** only one result (as a value), with same efficiency problems.

Call-by-Reference (i.e., l-value)

• Pass the **address** (l-value) of each actual parameter.

• On entry, the formal is bound to the address, which must be dereferenced to get value, but can also be **updated**.

• If actual argument doesn't have an l-value (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.,

```
PROCEDURE matmult(a,b,c: MATRIX)
... (* sets c := a * b *)

matmult(a,b,a) (* oops! *)
```

• **Call-by-value-result** (a.k.a. **copy-restore**) addresses this problem, but has other drawbacks.

Hybrid Methods

• Pascal, Ada, and similar languages allow the programmer to specify which arguments are to be handled as call-by-value, and which as call-by-reference. E.g., in Pascal, args are cbv by default, and cbr if the `VAR` keyword is used.

• In C, programmers can take the l-value of a variable explicitly, and pass that to obtain cbr-like behavior:

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

• C++ combines both these options:

```
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!

Records and Arrays

To understand argument passing of record and array types, must know what the language considers an r-value of these types to be!

• In Pascal, r-values of both arrays and records are the actual contents. So passing a record or array by value means copying the contents, whereas passing by reference (VAR parameter) doesn't.

• In PCAT, r-values of both records and arrays are **pointers** to the actual contents. PCAT has only call-by-value, but this doesn't actually cause copying, even for record or array values.

• In ANSI C/C++, `struct` r-values are the actual contents, but array r-values are pointers to the contents.

In this example, no doubles are copied on call:

```
typedef double vector[10];
double dotp(vector v, vector w) {
   double d = 0.0; int i;
   for (i = 0; i < 10; i++)
     d += v[i] * w[i];
   return d;
}
vector v1,v2;
double d1 = dotp(v1,v2);
```


Records and Arrays (continued)

To avoid copying C structures, must use pointers:

```
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 d1 = dotp(&v1,&v2);
```

• These issue also affect assignment, of course.

Substitution

• Can often get the effect we want using **substitution**, i.e., **macro-expansion**, e.g (in 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.,

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

expands to

```
{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" = Algol-60's **call-by-Name** facility.

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

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

Procedures as Parameters

It can be handy to pass **procedures** as parameters to other procedures. This feature is supported by Pascal, etc. and by C.

Example (pseudo-PCAT)

```
TYPE INTLIST IS RECORD x: INTEGER;
                       next: INTLIST;
                END;
TYPE INT_ACTION IS PROCEDURE(VAR i:INTEGER);
PROCEDURE do_intlist(a:INTLIST,f:INT_ACTION) IS
BEGIN
  IF a <> NULL THEN BEGIN
    f(a.x);
    do_intlist(a.next,f);
  END;
END;

PROCEDURE print_int(VAR i:INTEGER) IS
BEGIN WRITE(i); END;

var sum : INTEGER := 0;
PROCEDURE sum_int (VAR i: INTEGER) IS
BEGIN sum := sum + i; END;

VAR a: INTLIST;
do_intlist(a, print_int);
do_intlist(a, sum_int);
WRITE(sum);
```


Same Example in C

```
typedef struct intlist {
  int x;
  struct intlist *next;
} *Intlist;

typedef void (*int_action)(int);

void do_intlist(Intlist a, int_action f) {
  while (a) {
   (*f)(a->x);  /* or just f(a->x) */
   a = a->next;
  }
}

void print_int(int i) { printf("%d\n",i); }
int sum = 0;
void sum_int(int i) { sum += i; }
.
.
.
Intlist a;
do_intlist(a, &print_int);
               /* or just print_int */
do_intlist(a, &sum_int);
printf("%d\n", sum);
```


Using Local (Nested) Procedures

• Sometimes want to pass **local** functions as parameters.

Example: Improved version of sum:

```
PROCEDURE sum_list(a:INTLIST) : INTEGER IS
  VAR sum := 0;
  PROCEDURE sum_int(VAR i:INTEGER) IS
    BEGIN
      sum := sum + i;
    END;
  BEGIN
    do_intlist(a,sum_int);
    RETURN sum;
  END;
VAR c: INTLIST;
...WRITE(sum_list(c));...
```

• Here `sum_int` operates on the value of variable `sum`, which is neither local nor global.

• Solution: pass **pair** of (code-pointer,static-link) as "**value**" of procedure.

• Must guarantee that static link is still valid when procedure is called!

• Cannot express this in C.

More Nested Procedures

Example: Use iterator to count how many times specified integer occurs.

```
PROCEDURE count(i:INTEGER;a:INTLIST) IS
  VAR sum := 0;
  PROCEDURE check_int(VAR j:INTEGER) IS
    BEGIN
      IF j = i THEN sum := sum + 1; END;
    END;
  BEGIN
    do_intlist(a,check_int);
    RETURN sum;
  END;
VAR c: INTLIST;
...count(17,c);...
```

• Here `check_int` depends on the value of variable `i`, which is neither local nor global.

• Going one step further, can be handy to treat **procedure values** just like other values, e.g., to **return** them as function results or **store** them into variables.

"First-class" Procedures Example

```
TYPE COUNTER IS PROCEDURE(b:INTLIST) : INTEGER;
PROCEDURE make_counter(i:INTEGER) : COUNTER IS
  PROCEDURE count(a:INTLIST) IS
    VAR sum := 0;
    PROCEDURE check_int(VAR j: INTEGER) IS
      BEGIN
        IF j = i THEN sum := sum + 1; END;
      END;
    BEGIN
      do_intlist(a,check_int);
      RETURN sum;
    END;
  BEGIN
    RETURN count; (* !! *)
  END;
PROCEDURE main IS
  VAR g : COUNTER;
  VAR c,d: INTLIST;
  BEGIN
    g = make_counter(17);
    ...
    WRITE("c has ", g(c), "occurrences of 17");
    WRITE("d has ", g(d), "occurrences of 17");
  END;
```


Applying First-class Procedures

Example: Table-driven Command Processor

```
TYPE COMMAND IS ENUMERATION
      (SUMALL = 0, COUNT42S, COUNT101S, ...)

VAR actions: ARRAY OF COUNTER :=
   [<sum_list,
     make_counter(42),
     make_counter(101),
     ... >]

PROCEDURE dispatch(command:COMMAND, a: INTLIST)
IS
BEGIN
   WRITE ("Answer is ", actions[command] (a));
END
```

• Often handy for "call-backs" from operating system or window system.

Problems with first-class procedures

Consider activation tree for **make_counter** example:

```
                    main
                     /\
                    /  \
                   /    \
 make_counter(17)   g(c) == count(c)
                           |
                           |
                    do_intlist(c,check_int)
                           |
                           |
(requires value i = 17) check_int(j)
```

Activation of `make_counter` is no longer live when `count` is called!

If `i` is stored in activation record for `make_counter` and activation-record is stack-allocated, it will be gone at the point where `check_int` needs it!

To avoid this problem:

• Pascal prohibits "upward funargs;" procedure values can only be passed downward, and can't be stored.

• Some other languages only permit "top-level" procedures to be manipulated as procedure values (in C, this means **all** procedures!).

Heap Storage for Procedure Values

• Languages supporting first-class nested procedures (e.g., Lisp, Scheme, ML, Haskell, etc.) solve problem by using heap to store variables like `i`.

• Simple solution: Just put all activation records in the heap to begin with! (Garbage collection is a must!)

• More refined solution: Represent procedure values by a heap-allocated "closure" record, containing the procedure's code pointer and values of the non-local variables referenced by the procedure.

• Involves taking **copies** of the values of non-local variables, so only works when values are **immutable**.

• Can always introduce extra level of indirection to achieve this.

Functional Programming

What does **functional** mean? Two main senses:

• Programs consist of functions with **no side effects**.

• Functions are supported as **"first-class" values**.

Claim: functional programs are:

• clearer;

• easier to get right;

• easier to test;

• easier to transform;

• easier to parallelize;

• easier to prove things about.

Important examples:

• Lisp, Scheme ("strict", dynamically typed, impure)

• Standard ML, CAML ("strict", statically typed, impure)

• Haskell, Gofer ("lazy", statically typed, pure)

ML Example: Dictionaries with Sorted Lists

```
- type dict = int*string list;
- fun insert (d:dict) (k:int,v:string) : dict =
  case d of
    nil => [(k,v)]
  | (hk,hv)::t => if hk < k then
                       (hk,hv)::(insert t (k,v))
                  else if hk = k then
                         (k,v)::t
                  else (k,v)::d;

- val a = insert nil (1,"a");
val a = [(1,"a")] : dict
- val b = insert a (7,"g");
val b = [(1,"a"),(7,"g")] : dict
- val c = insert b (4,"d");
val c = [(1,"a"),(4,"d"),(7,"g")] : dict
- val d = insert b (9,"i");
val d = [(1,"a"),(7,"g"),(9,"i")] : dict

- fun member (d: dict) (k:int) : bool =
  case d of
    nil => false
  | (hk,_)::t => (hk = k) orelse
                 (hk < k andalso member t k);

- val q = member b 4;
val q = false : bool
```


Features

• Identifiers denote values, not changeable variables.

• `insert` is a **function** that returns a **new** list **without** changing its argument!

• Control structure is **recursion**, not iteration.

• Definition of lists is built into language; other similar data structures can be user-defined.

• List nodes (and other data structures) are tested and destructured using **pattern matching**.

Why bother?

Functions can't have side-effects. Therefore, they can't have dangerous, hidden side-effects!

Consider this C fragment:

```
insert(1,"q",t);
x = f(t); (* source not here... *)
s = member(1,t);
```

Will `s` be `true`? It depends whether `f` modifies `t`!

Compare this functional code:

```
val t' = insert (1,"q") t
val x = f t'
val s = member 1 t'
```

Here `s` must be `true`, because `f` **can't** modify it's argument (or anything else)! Also compare:

```
val t' = insert (1,"q") t
val (x,t'') = f t'
val s = member 1 t''
```


It's testable.

- True functions can always be **tested separately**.

```
fun appendx k t  =
  let val v = find k t
      val t' = delete k t
  in insert (k,v ^ "x") t'
  end
```

If `appendx` is wrong, then its definition is wrong, or `find` or `delete` or `insert` must be wrong.

The problem **can't** be due to a hidden interaction between `find`, `delete`, and/or `insert`.

Any coupling between functions must be made explicit in their arguments or return values.

This helps discourage coupling!

(In principle, debugging by divide and conquer can even be automated. In practice, conventional trouble-shooting is much easier.)