Parameter Passing, Calls, Symbol Tables & IRs

CS2210
Lecture 10

Parameter Passing

- Three semantic classes (semantic models) of parameters
  - IN: pass value to subprogram
  - OUT: pass value back to caller
  - INOUT: pass value in and back

- Implementation alternatives
  - Copy value
  - Pass an access path (e.g. a pointer)

Parameter Passing Methods

- Pass-by-Value
- Pass-by-Reference
- Pass-by-Result
- Pass-by-Value-Result (copy-restore)
- Pass-by-Name
Pass-by-value

- Copy actual into formal
  - Default in many imperative languages
  - Only kind used in C and Java
  - Used for **IN** parameter passing
  - Actual can typically be arbitrary expression including constant & variable

Pass-by-value cont.

- Advantage
  - Cannot modify actuals
    - So IN is automatically enforced

- Disadvantage
  - Copying of large objects is expensive
    - Don't want to copy whole array each call

- Implementation
  - Formal allocated on stack like a local variable
  - Value initialized with actual
  - Optimization sometimes possible: keep only in register

Pass-by-result

- Used for **OUT** parameters
  - No value transmitted to subprogram
  - Actual MUST be variable (more precisely lvalue)
    - foo(x) and foo(a[1]) are fine but not foo(3) or foo(x * y)
Pass-by-result gotchas

```c
procedure foo(out int x, out int y)
    g := 4;
    x := 42;
    y := 0;
} main() {
    b: array[1..10] of integer;
    g: integer;
    g = 0;
    call to foo;
}
```

- foo(a,a); print(a) what is printed?
- foo(b[g], ...): which element is modified?

Pass-by-value-result

- Implementation model for in-out parameters
- Simply a combination of pass by value and pass by result
  - Same advantages & disadvantages
  - Actual must be an lvalue

Pass-by-reference

- Also implements IN-OUT
  - Pass an access path, no copy is performed
- Advantages:
  - Efficient, no copying, no extra space
- Disadvantages
  - Parameter access usually slower (via indirection)
  - If only IN is required, may change value inadvertently
  - Creates aliases
Pass-by-reference aliases

```c
int g;
void foo(int& x) {
    x = 1;
}
foo(g);
```

- `g` and `x` are aliased

---

Pass-by-name

- Textual substitution of argument in subprogram
  - Used in Algol for in-out parameters, C macro preprocessor
  - Evaluated at each reference to formal parameter in subprogram
  - Subprogram can change values of variables used in argument expression
  - Programmer must rename variables in subprogram in case of name clashes
  - Evaluation uses reference environment of caller

---

Jensen’s device

```c
real procedure sigma(x, i, n);
value n;
real x, integer i, n;
begin
    real s;
    s := 0;
    for i := 1 step 1 until n do
        s := s + x;
    sigma := s;
end
```

What does `sigma(a(i), i, 10)` do?
Design Issues

- Typechecking
  - Are procedural parameters included?
  - May not be possible in independent compilation
  - Type-loophole in original Pascal: was not checked, but later procedure type required in formal declaration

Pass-by-name Safety Problem

```pascal
procedure swap(a, b);
  integer a,b,temp;
  begin
    temp := a;
    a := b;
    b := temp;
  end;
  swap(x,y):
   swap(i, x(i))
```

Call-by-name Implementation

- Variables & constants easy
- Reference & copy
- Expressions are harder
  - Have to use parameterless procedures aka. Thunks
**Thunk Example**

```plaintext
real procedure sigma(x, i, n);
value n;
real x; integer i, n;
begin
  real s;
  s := 0;
  for i := 1 step 1 until n do 
    s := s + x;
  sigma := s;
end
```

```plaintext
real procedure sigma(x|Thunk|(), i, 10);
begin
  Thunk := ADDRESS(I)
  function Thunk() :IntVarAddress;
  var exp;
  begin
    exp := a|Thunk|();
  end;
end;
```

```plaintext
function iThunk() :RealVarAddress;
var expi;
begin
  expi := a(I)*b(I);
  xiThunk := ADDRESS(expi);
end;
```

**Thunk Evaluation**

```plaintext
real procedure sigma(x|Thunk|(), i|Thunk|(), 10);
begin
  real s;
  s := 0;
  for i|Thunk|() := 1 step 1 until n do 
    s := s + xiThunk|Thunk|();
  sigma := s;
end
```

**Procedures as Parameters**

- In some languages procedures are first-class citizens, i.e., they can be assigned to variables, passed as arguments like any other data types
- Even C, C++, Pascal have some (limited) support for procedural parameters
- Major use: can write more general procedures, e.g. standard library in C:
  ```c
  qsort(void* base, size_t nmemb, size_t size, int (*compar)(const void*, const void*));
  ```
Design Issues

- Typechecking
  - Are procedural parameters included?
  - May not be possible in independent compilation
  - Type-loophole in original Pascal: was not checked, but later procedure type required in formal declaration

Procedures as parameters

- How do we implement static scope rules?
- = how do we set up the static link?

program parameters(input, output);

procedure b(function h(n : integer):integer);
begin writeln(h(2)) end {b};

procedure c;
var m : integer;
function f(n : integer) : integer;
begin f := m * n end {f};
begin m := 0; b(f) end {c};
begin c
end.
Solution: pass static link:

```
param
cb
```

Another Example

```
program fun;
procedure p1;
begin {...} end; {p1}

procedure p2(procedure x);
var n : integer;
 procedure p3;
begin n = 0; end; {p3}
begin x; p2(p3); end; {p2}
begin {main} p2(p1) end. {main}
```

Activation records

```
fun

p2

p1 p2

p3
```
**Procedure Calls**

- 5 Steps during procedure invocation
  - Procedure call (caller)
  - Procedure prologue (callee)
  - Procedure execution (callee)
  - Procedure epilogue (callee)
  - Caller restores execution environment and receives return value (caller)

**The Call**

- Steps during procedure invocation
  - Each argument is evaluated and put in corresponding register or stack location
  - Address of called procedure is determined
    - In most cases already known at compile / link time
  - Caller-saved registers in use are saved in memory (on the stack)
  - Static link is computed
  - Return address is saved in a register and branch to callee's code is executed

**The Prologue**

- Save fp, fp := sp , sp = sp - framesize
- Callee-saved registers used by callee are saved in memory
- Construct display (if used in lieu of static link)
**The Epilogue**
- Callee-saved registers that were saved are restored
- Restore old sp and fp
- Put return value in return register / stack location
- Branch to return address

**Post Return**
- Caller restores caller-saved registers that were saved
- Return value is used
- Division of caller-saved vs callee-saved is important
  - Reduces number of register saves
  - 4 classes: caller-saved, callee-saved, temporary and dedicated
  - Best division is program dependent so calling convention is a compromise

**Argument Registers**
- Additional register class used when many GPRs available
- Separate for integer and floating point arguments
- Additional arguments passed on stack
  - Access via fp+offset
Return values

- Return value register or memory if too large
- Could be allocated in caller’s or callee’s space
  - Callee’s space: not reentrant!
  - Caller’s space
    - Pass pointer to caller’s return value space
    - If size is provided as well callee can check for fit

Procedure Calls with Register Windows

- Register windows = hardware mechanism to facilitate parameter passing
  - First used on Sparc
  - Now also on Intel IA-64 (Itanium)
- Reduces addressing bits
  - Can usually provide larger register file
  - Caller’s out registers become callee’s in registers
  - Locals of caller not visible in callee

Sparc Register Windows
Symbol Tables
- Maps symbol names to attributes
  - Name: String
  - Class: Enumeration (storage class)
  - Volatile: Boolean
  - Size: Integer
  - Boundary: Integer
  - Bitsize: Integer
  - Bitbdry: Integer
  - Type: Enumeration or Type referent
  - BaseType: Enumeration or Type referent
  - Machtype: Enumeration
  - Neibs: Integer
  - Reg: String (register name)
  - Basereg: String
  - Disp: Integer (offset)

Symbol Table Operations
- New_Sym_Tab: SymTab -> SymTab
- Dest_Sym_Tab: SymTab -> SymTab
  - Destroys symtab and returns parent
- Insert_Sym: SymTab X Symbol -> boolean
  - Returns false if already present, otherwise inserts and returns true
- Locate_Sym: SymTab X Symbol -> boolean
- Get_Sym_Attr: SymTab X Symbol x Attr -> Value
- Set_Sym_Attr: SymTab X Symbol x Attr x Value -> boolean
- Next_Sym: SymTab X Symbol -> Symbol
- More_Syms: SymTab X Symbol -> boolean

Implementation Goals
- Fast insertion and lookup operations for symbols and attributes
- Alternatives
  - Balanced binary tree
  - Hash table (the usual choice)
    - Open addressing or
    - Buckets (commonly used)
Scoping and Symbol Tables

- Nested scopes (e.g., Pascal) can be represented as a tree
- Implement by pushing/pop symbol tables on/off a symbol table stack
- More efficient implementation with two stacks

Scoping with Two Stacks

Visibility versus Scope

- So far we have assumed scope \sim visibility
  - Visibility directly corresponds to scope this is called open scope
  - Closed scope = visibility explicitly specified
- Closed scope arises in OOPs (import and inheritance mechanisms)
- Can be implemented by adding a list of scope level numbers in which a symbol is visible
- Optimized implementation needs just one scope number
  - Stack represents declared scope or outermost exported scope
  - Hash table implements visibility by reordering hash chain
- Details in Muchnick 3.4
Intermediate Representations

- Make optimizer independent of source and target language
- Usually multiple levels
  - HIR = high level encodes source language semantics
  - Can express language-specific optimizations
  - MIR = representation for multiple source and target languages
  - Can express source/target independent optimizations
  - LIR = low level representation with many specifics to target
  - Can express target-specific optimizations

IR Goals

- Primary goals
  - Easy & effective analysis
    - Few cases
    - Support for things of interest
  - Easy transformations
    - General across source / target languages
- Secondary goals
  - Compact in memory
  - Easy to translate from / to
  - Debugging support
  - Extensible & displayable

High-Level IRs

- Abstract syntax tree + symbol table
  - most common
- LISP S-expressions
**Medium-level IRs**

- Represent source variables + temporaries and registers
- Reduce control flow to conditional + unconditional branches
- Explicit operations for procedure calls and block structure
- Most popular: three address code
  - \( t_1 := t_2 \text{ op } t_3 \) (address at most 3 operands)
  - if \( t \) goto \( L \)
  - \( t_1 := t_2 < t_3 \)

**Important MIRs**

- SSA = static single assignment form
  - Like 3-address code but every variable has exactly one reaching definition
  - Makes variables independent of the locations they are in
  - Makes many optimization algorithms more effective

**SSA Example**

```
x := u
...
... x ...
x := v
...
... x ...
```

```
x1 := u
...
... x1 ...
x2 := v
...
... x2 ...
```
Other Representations

- **Triples**
  1. \( i + 1 \)
  2. \( i := (1) \)
  3. \( p + 4 \)
  4. \( *{4} \)
  5. \( p := (4) \)
- **Trees**
  - Like AST but at lower level
  - Directed Acyclic Graphs (DAGs)
  - More compact than trees through node sharing

Three Address Code Example

for \( i := 1 \) to \( 10 \) do
  \( a[i] := b[i] + 5; \)
end

\( t_1 := i*4 \)
\( t_2 := 6b \)
\( t_3 := *(t_2 + t_1) \)
\( t_4 := t_3 + 5 \)
\( t_5 := i*4 \)
\( t_6 := 6a \)
\( *(t_6 + t_5) := t_4 \)
\( i := i + 1 \)

Representation Components

- **Operations**
- **Dependences between operations**
  - Control dependences: sequencing of operations
    - Evaluation of then & else depend on result of test
  - Data dependences: flow of values from definitions to uses
    - Operands computed before operation
    - Values read from variable before being overwritten
- **Want to represent only relevant dependences**
  - Dependences constrain operations, so the fewer the better
Representing Control Dependence

- Implicit in AST
  - Explicit as Control Flow Graphs (CFGs)
    - Nodes are basic blocks
      - Instructions in block sequence side effects
    - Edges represent branches (control flow between blocks)
  - Fanier:
    - Control Dependence Graph
      - Part of the PDG (program dependence graph)
    - Value dependence graph (VDG)
      - Control dependence converted to data dependence

Data Dependence Kinds

- True (flow) dependence (read after write RAW)
  - Reflects real data flow, operands to operation
- Anti-dependence (WAR)
- Output dependence (WAW)
  - Reflects overwriting of memory, not real data flow
    - Can often be eliminated

Data Dependence Example

```plaintext
x := 3
if q != NULL then
  y := x + 2
  w := *q
  x := z * 10
else
  x := 4
endif
```

- (1) x := 3
- (2) q != 0?
- (3) y := x + 2
- (4) w := *q
- (5) x := z * 10
- (6) x := 4
Representing Data Dependences (within bb’s)

- Sequence of instructions
  - Simple
  - Easy analysis
  - But: may overconstrain operation order

- Expression tree / DAG
  - Directly captures dependences in block
  - Supports local CSE (common subexpression elimination)
  - Can be compact
  - Harder to analyze & transform
  - Eventually has to be linearized

Representing Data Dependences (across blocks)

- Implicit via def-use
  - Simple
  - Makes analysis slow (have to compute dependences each time)

- Explicit: def-use chains
  - Fast
  - Space-consuming
  - Has to be updated after transformations

- Advanced options:
  - SSA
  - VDGs
  - Dependence glow graphs (DFGs)