HW1: Henry Peteet

September 20, 2016

1 Binding (35 points)

1. Write a simple program in an imperative language that contains at least one each of the following types of instructions: **(1)** an assignment of the result of an arithmetic operation; **(2)** a copy assignment from pointer variable to pointer variable; **(3)** a load from a pointer variable; **(4)** a store to a pointer variable; **(5)** a procedure call **(5 points)**.

   Choose an initial state for your program. Give the chosen initial state, followed by each intermediate state of the program along a complete run. Each state should be depicted as a scoped context paired with a store over memory cells **(10 points)**.

```
CODE                        CONTEXT
void swap(int* x, int* y) {
// call the old context we had before we called the function "A"
                            CTX: [x:0x5, y:0x6] [A]
                            STR: [0x0:3, 0x1:7, 0x2:0x1, 0x3:0x0, 0x4:0x0, 0x5:0x0, 0x6:0x1]
    int tmp = *x;           CTX: [x:0x5, y:0x6, tmp:0x7] [A]
                            STR: [0x0:3, 0x1:7, 0x2:0x1, 0x3:0x0, 0x4:0x0, 0x5:0x0, 0x6:0x1, 0x7:3][A]
    *x = *y;                CTX: [x:0x5, y:0x6, tmp:0x7] [A]
                            STR: [0x0:7, 0x1:7, 0x2:0x1, 0x3:0x0, 0x4:0x0, 0x5:0x0, 0x6:0x1, 0x7:3]
    *y = tmp;               CTX: [x:0x5, y:0x6, tmp:0x7] [A]
                            STR: [0x0:7, 0x1:3, 0x2:0x1, 0x3:0x0, 0x4:0x0, 0x5:0x0, 0x6:0x1, 0x7:3]
}

int main()
    int a = 1 + 2;          CTX: [a:0x0]
                            STR: [0x0:3]
    int b = 7;              CTX: [a:0x0, b:0x1]
                            STR: [0x0:3, 0x1:7]
    // ptr swap
    int* pa = &a;           CTX: [a:0x0, b:0x1, pa:0x2]
                            STR: [0x0:3, 0x1:7, 0x2:0x0]
    int* pb = &b;           CTX: [a:0x0, b:0x1, pa:0x2, pb:0x3]
                            STR: [0x0:3, 0x1:7, 0x2:0x0, 0x3:0x1]
    int* tmp = pa;          CTX: [a:0x0, b:0x1, pa:0x2, pb:0x3, tmp:0x4]
                            STR: [0x0:3, 0x1:7, 0x2:0x0, 0x3:0x1, 0x4:0x0]
    pa = pb;                CTX: [a:0x0, b:0x1, pa:0x2, pb:0x3, tmp:0x4]
                            STR: [0x0:3, 0x1:7, 0x2:0x1, 0x3:0x1, 0x4:0x0]
    pb = tmp;               CTX: [a:0x0, b:0x1, pa:0x2, pb:0x3, tmp:0x4]
                            STR: [0x0:3, 0x1:7, 0x2:0x1, 0x3:0x0, 0x4:0x0]
    // value swap
    swap(&a, &b);           CTX: [a:0x0, b:0x1, pa:0x2, pb:0x3, tmp:0x4]
                            STR: [0x0:7, 0x1:3, 0x2:0x1, 0x3:0x0, 0x4:0x0]
    printf("a=%d,b=%d", a, b);
}
```

2. Give a program P that returns a value when executing using static scoping that is different from the value that it gives when executing using dynamic scoping **(10 points)**.

```
// Prints 2 with static scoping and 1 with dynamic scoping.

int v = 0;

int main() {
    a();
    b();
    print(v);
}

void a() {
    v++;
}

void b() {
    int v = 100;
    a();
}
```

Rename the variables in P to form a program P' that when executed with *dynamic* scoping, gives the same value as P executed with *static* scoping **(10 points)**.

```
// Prints 2 in both static and dynamic scoping.
int global_v = 0;

int main() {
    a();
    b();
    print(v);
}

void a() {
    global_v++;
}

void b() {
    int b_v = 100;
    a();
}
```

2 Types (55 points)

1. Consider a very simple imperative language that allows a program to define (1) struct types in which each field is an integer and (2) reference types in which the data type is a struct. Let the language have a simple type conversion rule that states that for struct types T_0 and T_1, T_1 can be converted to T_0 if each field name of T_1 is a field name of T_0.

```
// pair is convertible to triple under these rules.
struct triple {
    int x,y,z;
}
struct pair {
    int x,y;
}
```

Suppose that the language is extended with a type-conversion rule that declares that if struct type T_0 can be converted to struct type T_1 then type T_0-reference can be converted to type T_1-reference. Is the resulting language strongly typed (i.e., is every program that type-checks under these rules guaranteed to only perform defined operations on values at runtime)? If so, give an informal argument for type-safety. If not, give a simple program that type-checks and input on which the program performs an undefined operation **(20 points)**.

```
// because pair is convertible to triple, pair* can be converted to triple*
struct pair a = {1,2};
struct pair* pa = &a;
struct triple b = *((triple*)pa);
// b.z has undefined behavior.
printf("b.z == %d", b.z);
```

Replace the type-conversion rule proposed with the following. For struct types T_0 and T_1 such that T_0 can be converted to T_1, T_1-reference can be converted to T_0 reference. Is the resulting language strongly typed (i.e., is every program that type-checks under these rules guaranteed to only perform defined operations on values at runtime)? If so, give an informal argument for type-safety. If not, give a simple program that type-checks and input on which the program performs an undefined operation **(20 points)**.

Answer: This language will only perform defined operations at runtime. If we convert between two struct references the new reference will always be to a struct that has fewer fields. In other words if we can convert T_0* to T_1* then the set of members of T_1 is a subset of the members of T_0. This means that all fields will be defined.

2. Consider a program that declares a two-dimensional array of integers a. Give a fragment of a simple assembly program that prints the integer at index 3 in the array at index 2 in the multi-dimensional array **(5 points)**.

```
;; NOTE: Using a fictional/simplified ISA for clarity.
;; Assume we have a multidimensional array called "A".
;; Assume it has dimensions A[4][4] (indexes 0-3 in both directions).
;; Assume r0 holds the address of the first element of A.
load r1, 2(r0) ; A[2]
load r2, 3(r1) ; A[2][3]
;; r2 now holds the int at A[2][3]
print r2 ; Assuming we have a simple print function
```

Assume that the language semantics guarantee that any array of arrays that are of constant size is allocated in row-major order. Write a program fragment that prints the integer at index 3 in the array at index 2 in the multi-dimensional array, using fewer loads then the first program fragment that you provided **(10 points)**.

```
;; Same assumptions as above, but we have guaranteed row-major order,
;; and we know this is of constant size 4x4.
;; Assume the we denote A[row][col].
;; This means we can calculate a constant offset by (row * #cols) + col;
;; in our case A[2][3] becomes A + 2 * 4 + 3 == A + 11
load r1, 11(r0)
print r1
```

3 Control Flow (35 points)

1. Give three expressions that are referentially transparent.

```
// setup
int a = 1;
int b = 2;
int* pa = &a;
int* pb = &b;
int rt, nt; // ref-transparent and not-transparent
// right hand side is referentially transparent
rt = *a + 2 * 3;
rt = *a + *b * 100;
rt = ((float)(a + b)) / 2.00f;
```

Give three expressions that are not referentially transparent.

```
// right hand side is not transparent (easy in C)
nt = a++;
nt = 2 + (a = a + b);
nt = *pb--;
```

Give an expression that is referentially transparent but stores to pointer cells. Expressions are allowed to contain calls to functions that you define **(15 points)**.

```
//setup
int a = 1;
int* pa = &a;
// right hand side is referentially transparent
rt = ++(*pa) * --a;
```

2. Give a standard low-level list data type using pointers.

```
typedef struct node_t {
    int value;
    struct node_t* next;
} node;
```

Give a non-recursive implementation of a procedure that computes the length of a given list;

```
int iter_length(node* n) {
      int len = 0;
      while(n != NULL) {
          len++;
          n = n.next;
      }
      return len;
}
```

translate it to a recursive implementation, using the transformation given in class **(10 points)**.

```
typedef struct {
      int len;
      node* n;
} frame;

int rec_length(frame f) {
      if !(f.n != NULL) { // negated while condition.
          return f.len;  // body after while
      }
      f.len++;       // body inside while
      f.n = f.n.next;
      return rec_length(f);
}
```

Give a recursive implementation of a procedure that computes the length of a given list;

```
int rec_length(node* n) {
      if (n == NULL) {
          return 0;
      }
      int r = rec_length(n.next);
      r = r + 1;
      return r;
}
```

translate it to a non-recursive implementation, using the transformation given in class **(10 points)**.

```
// Assume I have a type stack_t;
typedef struct {
    int r;
      node* n;
      int pc;
} frame;

int iter_length(node* n) {
    stack_t s = NewStack();
    frame initial_frame;
    initial_frame.n = n;
    initial_frame.pc = 0;
    int return_val = 0; // we could keep this in the frame but I think this is easier.
    while(!empty(s)) {
          frame f;
          f, s = pop(s);
          switch(f.pc) {
             case 0:
                if(f.n == NULL){
                     return_value = 0;
                  } else {
                     f.pc = 1;
                     s = push(s,f);
                     f.n = f.n.next;
                     f.pc = 0;
                     s = push(s,f);
                  }
                  continue
              case 1:
                  f.r = return_value;
                  f.r = r + 1;
                  return value = f.r;
                  continue;
          }
    }
    return return_value;
}
```