

Concepts of Programming Languages (INFOTECH)

Summer Semester 2008
Prof. Plödereder, Stefan Staiger/Steffen Keul

Solutions to Assignment #1

Home page: <http://www.iste.uni-stuttgart.de/ps/Lehre/>

1 Type equivalence

- Which variables are structurally equivalent?

Structurally equivalent: $\{A, B, C, D, E, F\}, \{X, Y, Z\}$

- Which variables are name equivalent? (Some cases are treated differently in different languages. Which cases are that and what possibilities exist?)

Name equivalent: $\{D, E, F\}$

In some languages (but not Ada) also name equivalent: $\{B, C\}, \{Y, Z\}$
(problem: anonymous types: no name)

- Name advantages and disadvantages of the two concepts of type equivalence!

Advantages of name equivalence:

- easy to implement for the compiler writer (problem for structural equivalence: recursive types)
- different semantics of structurally equivalent types can be expressed. Compare:

```
type Complex is record
  Re, Im : Float;
end record;
```

```
type Interval is record
  Start, End : Float;
end record;
```

Name equivalence thus gives increased safety

Disadvantages of name equivalence:

- very strict, often declares types unexpectedly as not equivalent
- problem with anonymous types

2 Type conversions

We have the following code snippet:

```
a, b: real;
c: int;

c := a + b;
```

Assume that the variables have the values $a = 3.6$ and $b = 3.8$. Which value has c after the assignment in a language with implicit type conversion (from **real** to **int**) if

- the conversion is done by mathematical rounding and is done as early as possible,

$$c = \text{round}(a) + \text{round}(b) = 4 + 4 = 8$$

- the conversion is done by mathematical rounding and is done as late as possible,

$$c = \text{round}(a + b) = \text{round}(7.4) = 7$$

- the conversion is done by truncating decimal places and is done as early as possible,

$$c = \text{trunc}(a) + \text{trunc}(b) = 3 + 3 = 6$$

- the conversion is done by truncating decimal places and is done as late as possible?

$$c = \text{trunc}(a + b) = \text{trunc}(7.4) = 7$$

3 Record and array layout

procedure Test (A: Integer; B: Integer) is

```
I: Integer;
type Rec is record
  Flags: array (1..3) of Boolean;
  Arr: array (1..A) of Integer;
  Arr2: array (1..B) of Integer;
  Status: Boolean;
  Count: Integer;
end record;
RObj: Rec;
K: Integer;
```

begin ... end Test;

- At what time is the size of the variable `RObj` determined?

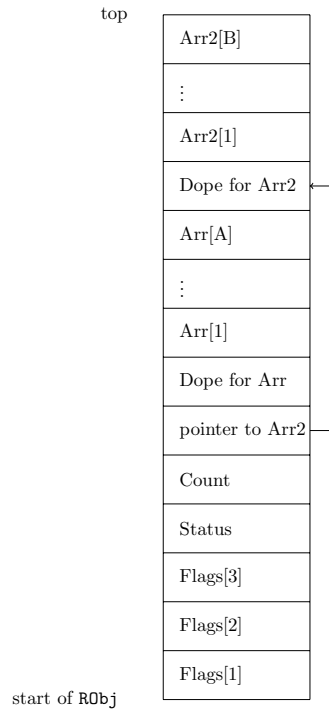
The size of `RObj` is determined when the procedure `Test` is called. It cannot be determined earlier since we need the values of `A` and `B`.

- Describe the layout of the record `RObj`. Choose one possible way to represent the arrays inside `RObj`. You are allowed to use different representations for each of the arrays.

- Rule: First come the components of statically known size to allow efficient access. Other components are placed at the end of the record.

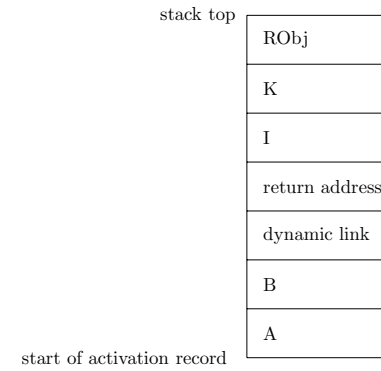
Note: Some languages don't allow such an reordering.

- We use padding to respect the machine's alignment restrictions.
- Booleans can be packed together in a single word, but remember that this might cost some additional instructions to read or modify one of them.
- Since the component size and number of elements for `Flags` is statically known, we can omit the dope for this array. On the other hand, for `Arr` and `Arr2`, we need a dope due to the unknown number of elements.
- The exact contents of the dope depend on the language features which must be supported. For example, the dope in Ada could contain lower and upper bound of the possible array indices, and perhaps also the size each component has ("stride").
- In our example below, the dope is stored directly in front of the array elements. We introduce an artificial pointer to the beginning of `Arr2`, although that address could also be found within the dope for `Arr`. This way, we save one add instruction for every access to a member of `Arr2`.
- Here's a possible layout of the record `RObj`:



- Describe the layout of the activation record (on the stack) in detail.

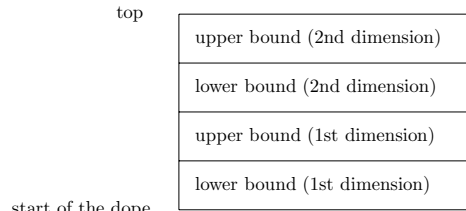
- The activation record typically contains the return address, the parameters and the local variables. Depending on the language, it sometimes also contains dynamic (and static) links to activation records of other procedures.
- The exact order in which the parameters are passed normally depends on the underlying system. In our example, we pass parameters left-to-right.
- Again, we arrange the components in such a way that all components of statically known size come first.
- The layout of `RObj` is as described above, so we don't repeat the details here.



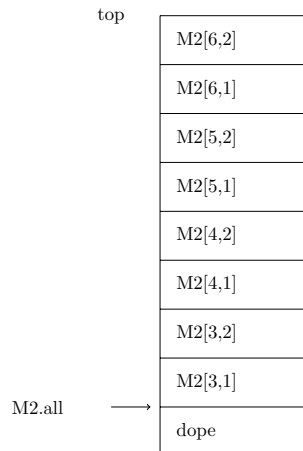
4 Multi-dimensional array

- Where are the contents of `M1` stored in procedure `Main`? On the stack or on the heap? Where are the contents of `M2` stored?
 - The contents of `M1` are stored on the stack, but the components of `M2` are placed on the heap because we used the allocator `new`. Note that the pointer to these heap locations will be stored on the stack.
- For the access to `M2` the array dope is needed. Why? Where could it be stored?
 - We need the dope to calculate the address at which we find the correct element. Since the type `Matrix` allows arguments to have a statically unknown size in the first dimension, we cannot calculate that address without the information passed in the dope.
 - The dope for `M2` should be stored on the heap along with the elements. It can be stored below the elements with the pointer `M2` actually pointing to the first element, thus having negative offsets to access the dope.
- What would happen if the line marked with (1) would be active and not commented out? And why?

- The access would be illegal since M1 has no index 1 in the second dimension. Using the dope, this can be detected at runtime. Then, an exception could be raised.
- Show the contents of the dope. Show how the elements of M2.a11 are arranged in memory, assuming row-major array layout. What would be the difference in column-major?
- The dope must tell us lower and upper bounds for both dimensions. The component size and the number of dimensions are statically known and thus we don't need these values in the dope. So the dope could look like this:



- row major: arrange the elements row by row, that is, first all elements of the first row, then all elements of the second row and so on. Difference for column-major: column by column
- Therefore, the elements of M2.a11 are arranged like this:



5 Recursive procedure

For the solution: The program executes in the following order:

1. Main calls Recursive (False)
2. Assignment 3 is executed

3. Control transfers back to Main
4. Main calls Recursive (True)
5. Assignment 1 is executed
6. Recursive calls Recursive (False)
7. Assignment 3 is executed
8. Control transfers back to the instance of Recursive with Condition = True
9. Assignment 2 is executed
10. Control tranfers back to Main, end of the program

- How many times is the procedure **Recursive** entered for one execution of the main program?

3 times.

In what order are the assignment statements executed during one execution of the whole program?

Order: 3 - 1 - 3 - 2.

- What value of X is used when assignment 2 is executed?

It uses the value 3 because X is a local variable (cannot be modified by other instances of Recursive).

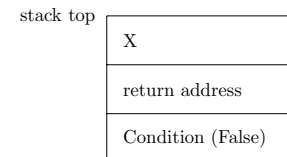
- What would happen if the call of the procedure **Recursive** to itself would pass the value **True** instead of **False**?

The procedure would call itself ever and ever again, without termination → program would "hang"

- List all the changes of the stack while the program executes. Include changes to variables and changes of the stack pointer.

- We describe the relative changes to the previous state and omit dynamic links:

1. *main program starts with the call to Recursive (False);*
The activation record for **Recursive** is added to the stack:



2. *assignment 3 is executed;*
The value 7 is stored in the location for X
3. *control is transfered back to the main program;*
The activation record created in step 1 is removed
4. *call to Recursive (True);*
A new activation record is created and added to the stack:

stack top

X
return address
Condition (True)

5. *assignment 1 is executed;*
The value 3 is stored in the location for X
6. *call to Recursive (False);*
A second activation record is added to the stack; thus the stack now looks like this:

stack top

X
return address
Condition (False)
X (3)
return address
Condition (True)

7. *assignment 3 is executed;*
The value 7 is stored in the top-most X which belongs to the current instance of **Recursive**
8. *control is transfered back to Recursive (True);*
The second activation record from step 6 is removed. The stack now looks like this:

stack top

X (3)
return address
Condition (True)

9. *assignment 2 is executed;*
has no impact on the stack (as far as we can tell here with the dots)
10. *control is transfered back to the main program;*
The activation record from step 4 is removed; thus, the stack now looks the same he did at the beginning.

- Show the call stack when assignment 2 is executed for the first time.
 - We can easily derive this from the previous observations:

stack top

X (3)
return address
Condition (True)