Assignment 7

CSCI 3136: Principles of Programming Languages

Due Apr 12, 2019

Submission instructions: Create a directory `a7` at the top level of your SVN repository. Place a single PDF file named `a7.pdf` containing your answers into this directory and submit it via `svn commit`.

Question 1 (10 marks) We discussed variadic functions in class. Recall that these are functions that can be called with a variable number of function arguments and are supported in many languages, including Java, C, and many more. Also recall that efficient access to function arguments and local variables of functions in statically scoped languages is achieved by deciding at compile time the offset relative to the frame pointer where each argument and variable is stored. This condition clearly cannot be preserved completely when the number of function arguments can vary. However, it can be preserved for all required function arguments and for local variables.

Propose an organization of the stack frame of a function call that meets the following requirements:

- All required function arguments can be found at a fixed offset from the frame pointer.
- All local function arguments can be found at a fixed offset from the frame pointer.
- The sequence of variable function arguments can be iterated over in C style (see the slides).
- The variable function arguments are stored directly in the stack frame, not in a separate array.
- The stack frame organization plays nicely with the standard function call sequence discussed in class: The caller pushes arguments and the return address onto the stack. The callee constructs the space for its local variables. Destruction of the stack frame happens in reverse.

Discuss briefly how to determine the offset of required function arguments and local variables relative to the frame pointer, given your proposed organization of the stack frame. Also describe the call sequence of calling a variadic function.

For this question, you may come up with your own solution or, provided that you cite the source, find information online on how C implements variadic functions.

Question 2 (5 marks) Whether two types are structurally equivalent is ultimately a matter of interpretation: the language designer decides which differences between two types are significant and which ones are not. On page 304 of the textbook, the following algorithm for determining structural equivalence of two types `A` and `B` is given:[1] Every type `T` can be represented as a string $s_{\mathtt{T}}$ defined inductively:

- If `T` is a built-in type, $s_{\mathtt{T}}$ is its name.
- If `T` is a composite of a sequence of subtypes $\mathtt{T}_1, \ldots, \mathtt{T}_k$, then $s_{\mathtt{T}}$ consists of the type constructor used to construct `T` followed by the strings $s_{\mathtt{T}_1}, \ldots, s_{\mathtt{T}_k}$.

For example, the string $s_{\mathtt{T}}$ for the following type `T` is "`struct char pointer int`":

```
struct T {
    char x;
    int *y;
};
```

[1]This is not a verbatim copy of the algorithm given in the textbook but a more rigorous inductive description that captures the same notion of structural type equivalence.

Two types A and B are structurally equivalent if and only if $s_A = s_B$.

(a) Argue why you cannot use this definition of type equivalence to decide whether the following two types A and B are equivalent:

```
struct A {
    char x;
    struct B *y;
};

struct B {
    char x;
    struct A *y;
};
```

(b) (10 bonus marks) Here is an alternative definition of type equivalence. Every composite type provides *element selectors* as mechanisms to access the parts it is composed of. For example, the type B defined in part (a) provides the element selectors .x and .y to access its constituent parts; an array type provides [] to access its elements. By applying element selectors recursively, we can define *valid sequences of element selectors*. For an object of type B in part (a), for example, the sequences .y.y.y.y and .y.y.y.x are valid and lead to objects of type B and char, respectively. The sequence .y.y.x.y is not valid because .y.y.x leads to an object of type char, which does not have a part accessible through the selector .y. Now we define that two types A and B are equivalent if they have the same (possibly infinite) set of valid sequences of element selectors and each such sequence that leads to a built-in type from A leads to the same built-in type from B and vice versa. It is not hard to see that, based on this definition, the two types A and B in part (a) are structurally equivalent because both have a set of valid sequences of element selectors consisting of the sequences .x, .y, .y.x, .y.y, .y.y.x, .y.y.y, ... and each such sequence ending in .x leads to an object of type char.

Provide an algorithm for testing whether two types are structurally equivalent using this definition. (While the set of valid sequences of element selectors of a type may be infinite, it can be represented using a finite structure and can be manipulated efficiently.) Argue that your algorithm is correct, that is, that it classifies two types as structurally equivalent if and only if they are according to the definition just given. (Hint: Try to model valid sequences of element selectors using a DFA and try to decide whether the two DFAs constructed for two types model the same language of type selectors.)