

Answer 7 of the 12 questions below.

1. Explain the differences between global, stack and heap variables. In particular, why do we need such a classification of variables and different implementation strategies for them? Comment upon the scopes as well as lifetime properties clearly bringing out the differences.
2. For this question, unless otherwise noted, we use the notation and definitions in the “Term Rewriting and *All That*” book. Prove or disprove the following, where (A, \rightarrow) is an abstract reduction system.
 - (a) If there exists some $a \in A$ such that the set $\{b \mid a \rightarrow b\}$ is infinite, then WFI (Well Founded Induction) does not hold for (A, \rightarrow) .
 - (b) If for all $a \in A$, the set $\{b \mid a \rightarrow^+ b\}$ is finite, then WFI holds for (A, \rightarrow) . (Recall that \rightarrow^+ is the transitive closure of \rightarrow .)
 - (c) If $\{b, a_0, a_1, a_2, \dots\} \subseteq A$, $b \rightarrow a_i$ for all $i \in \mathbb{N}$, and $a_0 \leftarrow a_1 \leftarrow a_2 \leftarrow \dots$, then WFI does not hold for (A, \rightarrow) . (Recall that \leftarrow is the inverse of \rightarrow .)
 - (d) Suppose that X is a finite, non-empty set and $(X, >)$ is a terminating reduction system. Let A be the set of finite sequences over X . The relation \rightarrow is the *least* relation satisfying the following conditions.
 - i. If $a \neq \langle \rangle$ (the empty sequence) then $a \rightarrow \langle \rangle$.
 - ii. If $a = x : b$ and $c = y : d$, then $a \rightarrow c$ holds iff $(x > y)$ or $(x = y$ and $b \rightarrow d)$. ($x : b$ denotes the sequence obtained by inserting x , an element of X , at the beginning of sequence b , an element of A .)
 WFI holds for (A, \rightarrow) .
3. Consider various manifestations of pattern matching in non-imperative programming languages. For example, Prolog uses unification to match queries to rules and facts. Dynamic dispatch (both single and multiple) in object-oriented languages can be thought of as a matching process between message calls and methods. This process can be further complicated by default parameters and implicit conversions. In functional languages, implicit function definitions enables calls to be bound to specific definitional instances. Type inferencing in these languages can also be thought of as matching variable occurrences to types.
 - (a) Describe the algorithms that perform these pattern matches and compare them with regard to their power and computational complexity.
 - (b) From the point of view of language design (that is, of the abstraction that a programmer sees), do you think that all of these mechanisms should be combined into a single pattern matching abstraction. Why or why not?

4. Which of the following CTL* and Mu-Calculus formulas can be expressed in LTL? In the case that there is an equivalent LTL formula, exhibit the formula and show that it is equivalent. Otherwise, prove that no LTL formula will do.

- (a) $AFAP$
- (b) $EAFGp$
- (c) $AFAGp$
- (d) $\neg[\nu Y(q \vee (p \wedge EXY))]$

5. Formulate and solve the following data-flow problem. Indicate whether it is a forward or a backward problem and what unification operator you are using. Also write the equations and an iterative algorithm with correct initializations that serve as a seed for the computation of fix point.

The Problem: We would like to catch uses of un-initialized variables and their effects in a program. Consider a quad : $a = b + c$. In this quad, if either b or c has an uninitialized reaching definition, then we term it as a use of un-initialized value. Moreover, it generates a value a in this case which has unpredictable behavior and its use further should also be termed as use of un-initialized value. This transitive closure property makes the problem interesting. We are thus interested in two sets : $IN[B]$ shows the definitions at the entry of basic block B which could carry un-initialized value, $OUT[B]$ shows the definitions at the exit of block B which could carry un-initialized value. Illustrate your solution through an example and briefly contrast your solution with reaching definitions analysis.

6. Static Single Assignment (SSA) and the Web are two most popular levels of representations used inside a compiler. Their chief goal is to provide a name-value separation that would promote their most efficient use in different phases of code generation. What is the difference between the SSA and Web intermediate forms (provide both formal and informal definitions) and show on a simple CFG how you will compute these two forms using some definitions of a variable. Indicate where (in which phase of code generation and optimization) and why one would use SSA and where and why one would use Web.

7. The completeness theorem of first-order logic establishes a connection between syntax and semantics. In 400 words or less, how would you describe this theorem, as well as the differences between syntax and semantics (in logic formalisms) to an undergraduate student?

8. Live Range Splitting is an important optimization as shown by Chow-Hennessey's famous work on priority based coloring for register allocation. What is the biggest benefit of splitting a live range? Give precise answer with proper illustration. If one is not careful in splitting a live range (for example, oversplitting), what are the problems that might arise? Chow-Hennessey proposed priority based splitting algorithm whose chief goal

is to preserve as longest a part of the live-range in a register as possible based on global priorities. Can you enlist a couple of weaknesses of this approach? Illustrate those shortcomings through small examples. What modifications do you suggest in overcoming these weaknesses? Illustrate your new algorithm on the same example and show your improvements.

9. Consider the following C routine. What function does it compute? (I.e., what is $\text{fun}(x)$ in relation to x ?) Provide appropriate preconditions and postconditions for the routine. Using an axiomatic semantics like Hoare logic, prove that this routine implements the function you have in mind.

```
int fun(int x) {
    int y1 = 0;
    int y2 = 0;
    int y3 = 0;

    while (y2 <= x) {
        y1++;
        y3 = y3 + 2;
        y2 = y2 + y3;
    }
    return y1;
}
```

10. A problem that programming language designers must confront is how to give meaning to recursive definitions (either functions or data types). As an example, consider the following situation.

```
define R {
    ...
    reference to R
    ...
}
```

The definition of R is not complete until the closing bracket is processed.

- (a) What mechanisms do the languages Lisp, C++, and Prolog provide the programmer to deal with recursion definitions? That is, what information can the programmer provide to let the language processor know that a recursive function/type is being defined?
- (b) How do you model mathematically a recursive definition in denotational semantics? How would you define a recursive function in the pure (untyped) lambda calculus?
11. Suppose we have a subtype relation $t1 <: t2$, meaning type $t1$ is a subtype of type $t2$. Example: $Int <: Rational <: Real$. Specify

- (a) A subtyping rule for the “tuple” type constructor $*$, allowing the conclusion $t1 * t2 <: t1' * t2'$
 - (b) A subtyping rule for the “function” type constructor $- >$, allowing the conclusion $t1 - > t2 <: t1' - > t2'$
12. Discuss the following questions regarding garbage collection.
- (a) Imagine that you want to develop a garbage collector for C++. What would be your design constraints? What decisions would you make? What are the potential problems with adding a garbage collector for general, unsuspecting C++ programs?
 - (b) Java is a garbage collected language. Nevertheless, many Java programs exhibit ever-growing memory consumption during their run time, indicating some kind of memory leak. Is this possible? Explain what could cause this phenomenon.