

Midterm Examination

15-312: Principles of Programming Languages

March 10, 2007

Name:

Andrew ID:

Instructions

- This exam is open-notes and open-book, but no computers are allowed.
- There are four problems, each worth 25%. You have 80 minutes to complete the exam.

| | Problem 1 | Problem 2 | Problem 3 | Problem 4 | Total | Extra credit |
|--------|-----------|-----------|-----------|-----------|-------|--------------|
| Score | | | | | | |
| Max | 25 | 25 | 25 | 25 | 100 | - |
| Grader | | | | | | |

1 Short Answers (5 Points Each)

1. True or False. Using our usual definition of the nat judgment and the following definition of plus , “typesaregreat” $\text{nat} \vdash \text{plus}(\text{zero}, \text{“typesaregreat”}, \text{“typesaregreat”})$ is derivable.

$$\frac{N \text{ nat}}{\text{plus}(\text{zero}, N, N)} \quad \frac{\text{plus}(N, M, P)}{\text{plus}(\text{succ}(N), M, \text{succ}(P))}$$

2. A database server supports transactions consisting of *init*, *read*, *write*, *commit*, *abort*, and *nest* messages. Each of these messages carries some data. Assume we have some type τ db which represents a database that holds elements of type τ . *init* messages takes a database, *read* messages take a database and a nat , *write* messages take a database, a τ , and a nat , *commit* messages takes a database and so do *abort* messages. Finally, *nest* messages keep track of a whole list of messages. Assuming you have some type τ list as in assignment 2, give a recursive labelled sum type for a fixed τ which describes these messages.

3. True or False. The principle of alpha-equivalence lets us define substitution without using the apartness judgment, $x \# y$.
4. State whether or not the following equivalence is valid under two different interpretations of PCF:

$$\text{ap}(\text{lam}[\tau](x.\text{zero}), e) \cong e : \tau$$

(a) Call-by-value:

(b) Call-by-name:

5. In PCF enriched with product types, give the value, if any, of the following expression:

$$\text{fst}(\text{rec}(x.\langle 0, x \rangle)).$$

Type Safety

In this problem you will prove safety for a simple stack based language THIRD- a primitive version of the real stack language FORTH. In THIRD, programs are a sequence of instructions that operate on a stack of natural numbers. A program stops executing when it runs out of instructions. For example, the add instruction removes the top two numbers from the stack and replaces them with their sum. We use the letters n, m, q, k to stand for natural numbers, p to represent programs, i for the class of instructions, and s for stacks. The complete syntax of THIRD is as follows:

$$\begin{array}{ll}
 \text{numbers} & n, m, q, k ::= \text{zero} \mid \text{succ}(n) \\
 \text{instructions} & i ::= \text{push}(n) \mid \text{pop} \mid \text{dup} \mid \text{add} \\
 \text{programs} & p ::= \cdot \mid i; p \\
 \text{stacks} & s ::= \cdot \mid n, s
 \end{array}$$

The instruction $\text{push}(n)$ pushes n onto the top of the stack, pop discards the top-most element of the stack, dup duplicates the top-most element of the stack, and add was previously described. The dynamic semantics is a four place relation, $p > s \mapsto p' > s'$, which says program p with stack s steps to program p' with stack s' . We define the dynamic semantics formally as follows:

$$\begin{array}{c}
 \frac{}{\text{push}(n); p > s \mapsto p > n, s} \quad \frac{}{\text{pop}; p > n, s \mapsto p > s} \\
 \frac{}{\text{dup}; p > n, s \mapsto p > n, n, s} \quad \frac{\text{plus}(n, m, q)}{\text{add}; p > n, m, s \mapsto p > q, s}
 \end{array}$$

The static semantics specify under what length of stack a program is valid. In this way we ensure that we never try to pop when the stack is empty. We say informally that $p : k$ holds if program p will be okay under a stack of at least length k . Assume we define the judgement n nat as usual. The $p : k$ judgement is defined as follows:

$$\begin{array}{c}
 \frac{k \text{ nat} \quad n \text{ nat} \quad p : \text{succ}(k)}{\cdot : k} \quad \frac{}{\text{push}(n); p : k} \quad \frac{p : k}{\text{pop}; p : \text{succ}(k)} \\
 \frac{p : \text{succ}(\text{succ}(k))}{\text{dup}; p : \text{succ}(k)} \quad \frac{p : \text{succ}(k)}{\text{add}; p : \text{succ}(\text{succ}(k))}
 \end{array}$$

For example, the dup static rule requires that the stack has length at least one to execute. On execution it adds an element to the stack resulting in a stack of at least length two. Thus, the remainder of the program needs to be safe to execute with a stack of at least length two.

In addition, the judgement $s : k$ states that the number of elements on the stack s is k .

Preservation If $p : k$ and $s : k$ and $p > s \mapsto p' > s'$ then there exists k' such that $p' : k'$ and $s' : k'$.

Progress If $p : k$ and $s : k$ then either $p = \cdot$ or there exists p' and s' such that $p > s \mapsto p' > s'$.

Problem 1 (10 points): Prove the case of progress for the pop instruction. You may use any inversion lemmas you find necessary without proof.

The proof is by case analysis on: _____.

Assume that $p : k$ and $s : k$. We consider here the case of the pop instruction.

From the static semantics we know: _____.

We wish to show: _____.

By the definition of stack typing, we know: _____.

Therefore, (complete the proof):

Problem 2 (10 points): Prove the case of preservation for pop. You may use any inversion lemmas you need without proof.

The proof is by case analysis on: _____.

Assume that $p > s \mapsto p' > s'$, where $p : k$ and $s : k$. We consider here the case of the pop instruction.

From the dynamic semantics we know: _____.

We wish to show: _____.

By inversion on program typing and stack typing, we know: _____.

Problem 3 (5 points): Explain in one sentence why the proofs of progress and preservation do not require an appeal to an inductive hypothesis, but are only case analyses.

Inductive Definitions

Regular expressions are commonly used in pattern matching. In this question you will consider an inductive definition of when a string matches a regular expression, and an extension of the concept of regular expression to encompass a simple form of context-free grammar.

We work with strings in an alphabet Σ , using ε to represent the empty string, a to represent a character in Σ , and $s_1 s_2$ to represent the concatenation of two strings s_1 and s_2 .

Below, we inductively define the judgement $r \sim s$, indicating when the regular expression r matches the string s exactly. (The Kleene star, or iteration, operator is deliberately omitted.)

$$\frac{}{\overline{a \sim a}} \quad \frac{}{\overline{\mathbf{1} \sim \varepsilon}} \quad \frac{}{\overline{\mathbf{0} \sim s}} \quad \frac{r_1 \sim s_1 \quad r_2 \sim s_2}{r_1 r_2 \sim s_1 s_2} \quad \frac{r_1 \sim s}{r_1 + r_2 \sim s} \quad \frac{r_2 \sim s}{r_1 + r_2 \sim s}$$

The iteration, r^* , of a regular expression r can be thought of as the solution to the recursion equation

$$r^* = \mathbf{1} + (r r^*).$$

This means that a string matches r^* iff it matches the right-hand side in the sense of the rules just given.

Problem 4 (10 points): Give *two rules* for matching r^* against a string based on the above equation:

The idea to consider r^* to be the solution of a recursion equation may be generalized to permit specification of simple forms of context-free grammar. Consider the following grammar for arithmetic expressions written in postfix notation (also known as “reverse polish” notation):

```
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
number ::= number digit | digit
rpn_calc ::= number
           | rpn_calc ; rpn_calc ; PLUS
           | rpn_calc ; rpn_calc ; TIMES
```

The grammar may be thought of as a recursion equation specifying that a string matches `rpn_calc` iff it matches one of the three clauses on the right-hand side. That is, we can think of `rpn_calc` as satisfying the recursion equation

$$\text{rpn_calc} = \text{number} + (\text{rpn_calc} ; \text{rpn_calc} ; \text{PLUS}) + (\text{rpn_calc} ; \text{rpn_calc} ; \text{TIMES}),$$

where we assume that `number` matches numbers and that `;` and `PLUS` and `TIMES` are “characters” of the alphabet.

Problem 5 (10 points): Introduce a new form of “regular expression” representing the solution to a generalized equation of the form just given for `rpn_calc`, and give the rule for matching a string against it. *Hint*: there is a close connection with a very similar concept in PCF.

Problem 6 (5 points): Using this new construct, give a definition of r^* .

Static Semantics

A Java procedure may specify a return value of an arbitrary type. For example, the header

```
Integer factorial (Integer x) ...
```

specifies a method that returns an `Integer` object, and

```
Float sqrt (Float x) ...
```

specifies a method that returns a `Float` object. Such a method can be used in a Java *expression* wherever a value of the result type is required.

Assume you are given a syntax for *expressions*, e , and a judgement $\Gamma \vdash e : \tau$, where τ is either `Integer`, `Float`, or `boolean`. The context Γ provides the types of the variables, as usual.

The body of a method is a *command*, c , defined by the following abstract syntax:

| | |
|-----------------------------|----------------------|
| $c ::= \text{assign}(x, e)$ | assignment $x := e$ |
| $\text{seq}(c_1, c_2)$ | sequence $c_1 ; c_2$ |
| $\text{if}(e, c_1, c_2)$ | conditional |
| $\text{return}(e)$ | return |

(Other forms of command are omitted for brevity.)

Problem 7 (10 points): Give an inductive definition of the judgement $\Gamma \vdash c : \tau$ stating that c is a command that returns a value of type τ , *if it returns at all*. Therefore, a command that does not execute a `return` statement is considered to have *any type*.

Problem 8 (15 points): Give an inductive definition of the judgement $\Gamma \vdash c ! \tau$ stating that the command c *must return* a value of type τ . *Hint*: your solution will involve the judgement $c : \tau$ just defined. Be sure to consider the case of sequencing carefully!