

# 15–312: Principles of Programming Languages

## MIDTERM EXAMINATION

March 5, 2009

- There are 13 pages in this examination, comprising 0 questions worth a total of 75 points.
- You have 80 minutes to complete this examination.
- Please answer all questions in the space provided with the question.
- There is a scratch sheet at the end for your use.
- You may refer to your personal notes and to the text, but to no other person or source, during the examination.

Full Name: \_\_\_\_\_

Andrew ID: \_\_\_\_\_

Short Answer[25]

Answer the following questions in the space provided. *Answers should take the form of a single, short sentence.* **Gödel's T**

Recall the language Gödel's **T** given Chapter 14 of *Practical Foundations*. Does the following dynamic semantics for function application specify call-by-value or call-by-name?

$$\frac{}{(\lambda x : \tau. e_1 \ e_2) \mapsto [e_2/x]e_1} \text{ app}$$

(a) (2 points)

(a) \_\_\_\_\_

[It is call-by-name.]

(b) (2 points) Are these two expressions  $\alpha$ -equivalent?

$\lambda z : \tau. \text{rec } z \{z \Rightarrow z \mid s(x) \text{ with } y \Rightarrow z\}$

$\lambda x : \tau. \text{rec } x \{z \Rightarrow z \mid s(z) \text{ with } y \Rightarrow x\}$

(b) \_\_\_\_\_

[Yes.]

## Plotkin's PCF

Recall the language PCF from Chapter 15 of *Practical Foundations*.

- (a) (3 points) Is there any input on which this function terminates? If so, please give one.

$$\text{fix } f : \text{nat} \rightarrow \text{nat} \text{ in } \lambda x : \text{nat} . \text{ifz } x \{z \Rightarrow (f \ x) \mid \text{s}(y) \Rightarrow (f \ x)\}$$

(a) \_\_\_\_\_

[No.]

- (b) (3 points) Assuming that  $\text{times} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$  is a correct implementation of multiplication over  $\text{nat}$ 's. What does this function do?

$$\text{fix } f : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \text{ in } \lambda x : \text{nat} . \lambda y : \text{nat} . \text{ifz } y \{z \Rightarrow \text{s}(z) \mid \text{s}(y') \Rightarrow ((\text{times } x) (f \ y'))\}$$

(b) \_\_\_\_\_

[It calculates  $x^y$ .]

## Recursive Types

Recall the extension of PCF with recursive types as in Chapter 19 of *Practical Foundations*. Consider the ML data type

$$\text{datatype regexp} = \text{Void} \mid \text{Cat of regexp} * \text{regexp} \mid \text{Star of regexp}$$

which has three constructors for creating values of this type.

- (a) (3 points) Give a type  $\tau_{\text{regexp}}$  corresponding to this ML data type using *only* sums, products, and recursive types.

(a) \_\_\_\_\_

$[\mu r . (\text{unit} + ((r \times r) + r))]$

- (b) (3 points) In terms of your solution for the preceding question, give the representation of the ML value constructor **Star**. Your solution should be an expression of type  $\tau_{\text{regexp}} \rightarrow \tau_{\text{regexp}}$ .

(b) \_\_\_\_\_

$[\lambda x : \tau_{\text{regexp}} . \text{fold}(\text{in}[r](\text{in}[r](x)))]$

## Polymorphism

Recall the language System F from Chapter 23 of *Practical Foundations*.

- (a) (3 points) Is it possible to define general recursion in System F? If so, give a definition; if not, explain why not.

**Solution:** No, because you can write total functions only.

- (b) (3 points) Give a Church-style representation of the ML datatype `regexp` specified above as a polymorphic type in System F.

Hint: Recall that we define these encodings in terms of “handlers” for each of the possible constructors of the type.

**Solution:**  $\forall t.t \rightarrow (t \times t \rightarrow t) \rightarrow (t \rightarrow t) \rightarrow t$ .

## Dynamic Typing

Recall dynamic typing from Chapter 22 of *Practical Foundations*.

- (a) (3 points) True or False: In a dynamically typed language types are attached to *values* at run-time, rather than to *expressions* at compile-time, as they are in a statically typed language.

(a) \_\_\_\_\_

[False.]

**Solution:** Dynamically typed languages are statically typed; the tags attached to values are not types.

## Inductive Def's

Recall the inductive definition of lists in Chapter 1 of *Practical Foundations*.

Let  $bs$  be a list of booleans,  $\text{tt}$  and  $\text{ff}$ , and let  $d$  be a “deck” of “cards” also represented as a list of objects  $c$ . (We are not concerned here with the representation of cards.) We write  $\text{nil}$  for the empty list (of booleans or cards), and  $\text{cons}(x; xs)$  for the list with head  $x$  and tail  $xs$ .

Below we give an inductive definition of a four-place judgement  $\text{split}(bs, d, d_1, d_2)$  stating that the deck,  $d$ , is to be split into two parts,  $d_1$  and  $d_2$ , according to the bits  $bs$ . Each bit determines whether the next card in  $d$ , if any, is to be placed onto  $d_1$  or  $d_2$ . If all bits are exhausted, all cards are distributed to  $d_1$ .

$\frac{}{\text{split}(\text{nil}, d, d, \text{nil})} \text{ no-bits}$
$\frac{}{\text{split}(\text{cons}(b; bs), \text{nil}, \text{nil}, \text{nil})} \text{ no-cards}$
$\frac{\text{split}(bs, d, d_1, d_2)}{\text{split}(\text{cons}(\text{tt}; bs), \text{cons}(c; d), \text{cons}(c; d_1), d_2)} \text{ choose-left}$
$\frac{\text{split}(bs, d, d_1, d_2)}{\text{split}(\text{cons}(\text{ff}; bs), \text{cons}(c; d), d_1, \text{cons}(c; d_2))} \text{ choose-right}$

You are to prove that  $\text{split}$  has the mode  $(\forall, \forall, \exists, \exists)$ . That is, you are to prove

For all  $bs$  and for all  $d$ , there exist  $d_1$  and  $d_2$  such that  $\text{split}(bs, d, d_1, d_2)$ .

The proof will be by induction on  $bs$ .

Use the following for the induction property:

$\mathcal{P}(bs) =$  “For all  $d$ , there exist  $d_1$  and  $d_2$  such that  $\text{split}(bs, d, d_1, d_2)$ ”

(a) (5 points) Case 1:  $bs = \text{nil}$

To show:  $\mathcal{P}(\text{nil}) =$  “For all  $d$ , there exist  $d_1$  and  $d_2$  such that  $\text{split}(\text{nil}, d, d_1, d_2)$ ”

Complete this case of the proof.

**Solution:** Let  $d$  be given.  
 Take  $d_1 = d$ .  
 Take  $d_2 = \text{nil}$ .  
 By no-bits,  $\text{split}(\text{nil}, d, d_1, d_2)$ .

(question continued on next page)

(b) Case 2:  $bs = \text{cons}(b; bs')$

Assume  $\mathcal{P}(bs') = \text{"For all } d', \text{ there exist } d_1 \text{ and } d_2 \text{ such that } \text{split}(bs', d', d_1, d_2)\text{"}$

To show:  $\mathcal{P}(\text{cons}(b; bs')) = \text{"For all } d, \text{ there exist } d_1 \text{ and } d_2 \text{ such that } \text{split}(\text{cons}(b; bs'), d, d_1, d_2)\text{"}$

Proceed by case analysis on  $d$ :

i. (5 points) Case 2.1:  $d = \text{nil}$

Suffices to show: There exist  $d_1$  and  $d_2$  such that  $\text{split}(\text{cons}(b; bs'), \text{nil}, d_1, d_2)$

Complete this case of the proof.

**Solution:** Take  $d_1 = d_2 = \text{nil}$ .  
By no-cards,  $\text{split}(bs, d, d_1, d_2)$ .

ii. Case 2.2:  $d = \text{cons}(c; d')$

Consider the possible values for  $b$ :

$\alpha$ ) (15 points) Case 2.2.1:  $b = \text{tt}$

Suffices to show: There exist  $d_1$  and  $d_2$  such that  $\text{split}(\text{cons}(\text{tt}; bs'), \text{cons}(c; d'), d_1, d_2)$

Complete this case of the proof.

**Solution:** By the induction hypothesis on  $bs'$  applied to  $d'$ , there exist  $d'_1$  and  $d'_2$  such that  $\text{split}(bs', d', d'_1, d'_2)$ .  
Take  $d_1 = \text{cons}(c; d'_1)$ .  
Take  $d_2 = d'_2$ .  
By choose-left,  $\text{split}(bs, d, d_1, d_2)$ .

$\beta$ ) Case 2.2.2:  $b = \text{ff}$

Suffices to show: There exist  $d_1$  and  $d_2$  such that  $\text{split}(\text{cons}(\text{ff}; bs'), \text{cons}(c; d'), d_1, d_2)$

This case is similar to the previous one; **YOU DO NOT NEED TO DO THIS CASE OF THE PROOF.**

**Solution:** By the induction hypothesis on  $bs'$  applied to  $d'$ , there exist  $d'_1$  and  $d'_2$  such that  $\text{split}(bs', d', d'_1, d'_2)$ .  
Take  $d_1 = d'_1$ .  
Take  $d_2 = \text{cons}(c; d'_2)$ .  
By choose-right,  $\text{split}(bs, d, d_1, d_2)$ .

## Type Safety

In this problem, we are going to extend Gödel's T with a type of *infinite streams of natural numbers*, and then you will only prove part of the progress theorem for this extension.

## The Type and Syntax of Streams

We begin by giving the syntax of the extension of Gödel's T with streams:

$$\begin{aligned}\tau & ::= \dots \mid \text{stream} \\ e & ::= \dots \mid \text{head}(e) \mid \text{tail}(e) \mid \text{generate from } e \text{ emit } x. e_1 \text{ next } y. e_2\end{aligned}$$

You may use the ABT form  $\text{gen}(e, x. e_1, y. e_2)$  instead of the concrete syntax in your answers.

## Static Semantics

We give the typing rules for infinite streams, which are given in the nested box below.

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e_1 : \text{nat} \quad \Gamma, y : \tau \vdash e_2 : \tau}{\Gamma \vdash \text{generate from } e \text{ emit } x. e_1 \text{ next } y. e_2 : \text{stream}} \text{ TGen}$$
$$\frac{\Gamma \vdash e : \text{stream}}{\Gamma \vdash \text{head}(e) : \text{nat}} \text{ THead} \qquad \frac{\Gamma \vdash e : \text{stream}}{\Gamma \vdash \text{tail}(e) : \text{stream}} \text{ TTail}$$

The typing rules for the head and tail are straightforward. The rule *TGen* is the typing rule for the introduction rule for the stream type, and gives the typing properties for the three parts of ( $\text{generate from } e \text{ emit } x. e_1 \text{ next } y. e_2$ ).

- The subterm  $e$  of type  $\tau$  is called the *seed* of the infinite stream. The seed provides data for the calculation of the head and tail of the stream, which are specified by the other two arguments of the generator expression.
- The subterm  $e_1$  has type  $\text{nat}$ , under the hypothesis that  $x$  has type  $\tau$ . The head of a stream can be computed by substituting the seed for  $x$  into  $e_1$ .
- The subterm  $e_2$  has type  $\tau$ , under the hypothesis that  $y$  has type  $\tau$ . By substituting  $e$  for  $y$  into  $e_2$ , we can compute the seed for the stream forming the tail of the current stream.

For example, we can construct various streams of natural numbers using the `generate` form in the following way:

$$\begin{aligned}\text{nats} & = \text{generate from } z \text{ emit } x. x \text{ next } y. s(y) \\ \text{evens} & = \text{generate from } z \text{ emit } x. x \times 2 \text{ next } y. s(y) \\ \text{odds} & = \text{generate from } z \text{ emit } x. s(x) \text{ next } y. s(s(y))\end{aligned}$$

## Dynamic Semantics

The value judgement and the transition judgement for stream expressions are given in the two boxes below.

$$\frac{}{\text{generate from } e \text{ emit } x. e_1 \text{ next } y. e_2 \text{ value}} \text{VGen}$$

A value of stream type is a generator expression (`generate from  $e$  emit  $x$ .  $e_1$  next  $y$ .  $e_2$` ). We can formally state this fact as the canonical forms property for the stream type.

**Lemma 1.** (*Canonical Forms for Streams*) *If  $\cdot \vdash e : \text{stream}$  and  $e$  value, then there exist  $e_0, e_1, e_2$  such that  $e = \text{generate from } e_0 \text{ emit } x. e_1 \text{ next } y. e_2$ .*

We take this as given, without showing the proof.

$$\frac{e \mapsto e'}{\text{head}(e) \mapsto \text{head}(e')} \text{EHeadArg}$$

$$\frac{}{\text{head}(\text{generate from } e \text{ emit } x. e_1 \text{ next } y. e_2) \mapsto [e/x]e_1} \text{EHead}$$

$$\frac{e \mapsto e'}{\text{tail}(e) \mapsto \text{tail}(e')} \text{ETailArg}$$

$$\frac{}{\text{tail}(\text{generate from } e \text{ emit } x. e_1 \text{ next } y. e_2) \mapsto \text{generate from } [e/y]e_2 \text{ emit } x. e_1 \text{ next } y. e_2} \text{ETail}$$

In rule *EHead*, we get the head of an infinite stream (`generate from  $e$  emit  $x$ .  $e_1$  next  $y$ .  $e_2$` ) by substituting the seed  $e$  into  $e_1$ . Likewise, in rule *ETail*, we compute the tail of an infinite stream (`generate from  $e$  emit  $x$ .  $e_1$  next  $y$ .  $e_2$` ), by updating the generator's seed with a new value derived from  $e_2$ . For example, for *nats*, the stream of natural numbers defined above, we have:

$$\text{tail}(\text{generate from } z \text{ emit } x. x \text{ next } y. s(y)) \mapsto \text{generate from } s(z) \text{ emit } x. x \text{ next } y. s(y)$$

The result clearly generates natural numbers starting from 1, instead of 0.

## Proving Progress

Now, we will prove the progress lemma for this language.

**Lemma 2.** (*Progress*) *If  $\Gamma \vdash e : \tau$ , then either  $e$  value or there exists  $e'$  such that  $e \mapsto e'$ .*

We use rule induction on typing derivations with the property  $\mathcal{P}(\Gamma \vdash e : \tau) = \Gamma \vdash e : \tau$  and (if  $\Gamma = \cdot$  then  $e$  value or there exists  $e'$  such that  $e \mapsto e'$ ).

Your task is to prove this theorem for the *THHead* case.

(a) In both of the following questions, fully expand any uses of  $\mathcal{P}$ .

- i. (2 points) First, you should explicitly state what you may assume as an induction hypothesis.

**Solution:** We may assume  $\Gamma \vdash e : \text{stream}$  and (if  $\Gamma = \cdot$  then  $e$  value or there exists  $e'$  such that  $e \mapsto e'$ ).

- ii. (3 points) Now, state what your proof obligation is – that is, state what you need to show.

**Solution:** We must show that  $\Gamma \vdash \text{head}(e) : \text{nat}$  and (if  $\Gamma = \cdot$  then  $\text{head}(e)$  value or there exists  $e'$  such that  $\text{head}(e) \mapsto e'$ ).

(b) Now, we have a partial proof outline, and your job is to turn it into a proper proof by filling in the missing details. You may freely use inversion lemmas, the substitution lemma, and the canonical forms lemma for each type, as long as you note which one you appeal to.

- (a) (2 points) Assume  $\mathcal{P}(\Gamma \vdash e : \text{stream})$ . This means that  $\Gamma \vdash e : \text{stream}$  and also that

(a) \_\_\_\_\_

[if  $\Gamma = \cdot$  then  $e$  value or exists  $e'$  s.t.  $e \mapsto e'$ ]

- (b) (2 points) To conclude that  $\Gamma \vdash \text{head}(e) : \text{nat}$ , what do we need to do?

(b) \_\_\_\_\_

[Apply rule *THHead* to (a)]

- (c) Now, assume  $\Gamma = \cdot$ .

- i. (2 points) How can we conclude  $e$  value or exists  $e''$  s.t.  $e \mapsto e''$ ?

i. \_\_\_\_\_

[Using the assumption  $\Gamma = \cdot$  and (a)]

- ii. Case analyzing i., we have two cases. (continued on next page)

- (a) (7 points) The first case is if  $e \mapsto e''$ . Now show  $\text{head}(e) \mapsto e'$  for some  $e'$ .

**Solution:**

By rule *EHeadArg*,  $\text{head}(e) \mapsto \text{head}(e'')$   
Take  $e'$  to be  $\text{head}(e'')$ , and  $\text{head}(e) \mapsto e'$

(b) (7 points) The second case is if  $e$  value. Now show  $\text{head}(e) \mapsto e'$  for some  $e'$ .

**Solution:**

By canonical forms,  $e = \text{generate from } e_0 \text{ emit } x. e_1 \text{ next } y. e_2$  for some  $e_0, e_1, e_2$

By rule *EHead*,  $\text{head}(\text{generate from } e_0 \text{ emit } x. e_1 \text{ next } y. e_2) \mapsto [e_0/x]e_1$

Take  $e'$  to be  $[e_0/x]e_1$ , and  $\text{head}(e) \mapsto e'$

## Language Design

Consider an abstraction of a relational database in which the type  $\mathbf{db}(\tau)$  represents a database whose *schema* is given by the type  $\tau$ . The schema may be any type at all, but in practice it is often a labelled product type of the form

$$\langle l_1 : \tau_1, l_2 : \tau_2, \dots, l_n : \tau_n \rangle.$$

The labels are the *columns*, or *attributes*, of the database, and the corresponding types describe the data in each column. For example,  $\langle \text{id} : \text{nat}, \text{salary} : \text{nat} \rangle$  might be the schema of a database with two columns.

The database is structured as a tree of *entries*, which are elements of the schema type. When the schema is a product type, the elements are labelled tuples, called *rows*. The introductory rules for the type  $\mathbf{db}(\tau)$  are as follows:

$$\frac{}{\Gamma \vdash \text{empty}_\tau : \mathbf{db}(\tau)} \text{empty} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{row}(e) : \mathbf{db}(\tau)} \text{row}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{db}(\tau) \quad \Gamma \vdash e_2 : \mathbf{db}(\tau)}{\Gamma \vdash \text{join}(e_1, e_2) : \mathbf{db}(\tau)} \text{join}$$

The eliminatory form for the type  $\mathbf{db}(\tau)$  is a recursor  $\mathbf{dbrec}(e, e_0, x. e_1, y.z. e_2)$  with the following static semantics:

$$\frac{\Gamma \vdash e : \mathbf{db}(\sigma) \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \sigma \vdash e_1 : \tau \quad \Gamma, y : \tau, z : \tau \vdash e_2 : \tau}{\Gamma \vdash \mathbf{dbrec}(e, e_0, x. e_1, y.z. e_2) : \tau} \text{dbrec}$$

The dynamic semantics of the recursor is specified by four rules, one to evaluate its argument and one for each introductory form.

$$\frac{e \mapsto e'}{\mathbf{dbrec}(e, e_0, x. e_1, y.z. e_2) \mapsto \mathbf{dbrec}(e', e_0, x. e_1, y.z. e_2)} \text{dbrec-arg}$$

$$\frac{}{\mathbf{dbrec}(\text{empty}_\tau, e_0, x. e_1, y.z. e_2) \mapsto e_0} \text{dbrec-emp}$$

$$\frac{}{\mathbf{dbrec}(\text{row}(e), e_0, x. e_1, y.z. e_2) \mapsto [e/x]e_1} \text{dbrec-row}$$

$$\frac{e'_3 = \mathbf{dbrec}(e_3, e_0, x. e_1, y.z. e_2) \quad e'_4 = \mathbf{dbrec}(e_4, e_0, x. e_1, y.z. e_2)}{\mathbf{dbrec}(\text{join}(e_3, e_4), e_0, x. e_1, y.z. e_2) \mapsto [e'_3, e'_4/y, z]e_2} \text{dbrec-join}$$

(The question continues on the next page.)

(a) (12 points) Using `dbrec` give a definition of the `project[li]` function of type

$$\text{db}(\langle l_1 : \tau_1, l_2 : \tau_2, \dots, l_n : \tau_n \rangle) \rightarrow \text{db}(\langle l_i : \tau_i \rangle)$$

that computes the projection of the database on the column labelled  $l_i$ .

**Solution:**

$$\lambda x : \tau_{\text{db}}. \text{dbrec}(x, \text{empty}_{\text{db}(\langle l_i : \tau_i \rangle)}, y. \text{row}(\langle l_i = y.l_i \rangle), u.v. \text{join}(u, v))$$

(b) (13 points) Using `dbrec` give a definition of the `select[li]` function of type

$$(\tau_i \rightarrow \text{bool}) \rightarrow \text{db}(\langle l_1 : \tau_1, l_2 : \tau_2, \dots, l_n : \tau_n \rangle) \rightarrow \text{db}(\langle l_1 : \tau_1, l_2 : \tau_2, \dots, l_n : \tau_n \rangle)$$

that selects those rows of the database whose  $i^{\text{th}}$  component satisfies the given predicate. You can use the standard elimination form for `bool`'s given by `if(e, etrue, efalse)`

**Solution:**

$$\lambda p : \tau_i \rightarrow \text{bool}. \lambda x : \tau_{\text{db}}. \text{dbrec}(x, \text{empty}_{\tau_{\text{db}}}, y. \text{if}(p \ y.l_i, \text{row}(y), \text{empty}_{\tau_{\text{db}}}), u.v. \text{join}(u, v))$$

**Scratch Work:**