

# 15–312: Principles of Programming Languages

## MIDTERM EXAMINATION

March 4, 2010

- There are 12 pages in this examination, comprising 3 questions worth a total of 100 points.
- You have 80 minutes to complete this examination.
- Please answer all questions in the space provided with the question.
- There a scratch page 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: \_\_\_\_\_

Question:	Short Answer	Type Safety	Loopy Lists	Total
Points:	35	35	30	100
Score:				



- (c) (5 points) Write the function  $isEven$  using primitive recursion.  $isEven$  is an expression of type  $nat \rightarrow nat$  that should evaluate to  $z$  if its input is even, and  $s(z)$  if its input is odd. Hint: It may be useful to define an expression  $not$  – also by primitive recursion – of type  $nat \rightarrow nat$ , such that  $not\ z = s(z)$ , and  $not\ s(z) = z$ .

**Solution:**  $not = \lambda x : nat.natrec\ x\ \{z \Rightarrow s(z) \mid s(-)\ \text{with}\ \_ \Rightarrow z\}$   
 $isEven = \lambda x : nat.natrec\ x\ \{z \Rightarrow z \mid s(-)\ \text{with}\ y \Rightarrow not\ y\}$

### $\alpha$ -equivalence

Are these following pairs of expressions  $\alpha$ -equivalent?

- (d) i. (3 points)

$$x + \lambda x.x$$

$$y + \lambda y.y$$

i. \_\_\_\_\_ **No.**

- ii. (3 points)

$$\lambda x.\lambda x.x$$

$$\lambda y.\lambda x.y$$

ii. \_\_\_\_\_ **No.**

- iii. (3 points)

$$\lambda x.\lambda y.x + y$$

$$\lambda x.\lambda y.y + x$$

iii. \_\_\_\_\_ **No.**

## Inductive Definitions

Assume we have the following judgements describing a file system. This is a simultaneous inductive definition, as it is defining both  $f$  file, and  $l$  listing together.

$$\frac{}{\text{empty listing}} TEmpty \quad \frac{s \text{ bitstring}}{\text{ord}(s) \text{ file}} TFile \quad \frac{d \text{ listing}}{\text{dir}(d) \text{ file}} TDir \quad \frac{f \text{ file} \quad d \text{ listing}}{(f, d) \text{ listing}} TListing$$

- (e) (6 points) What is the principle of rule induction for the above rules. That is, in order to show  $P(f \text{ file})$  whenever  $f \text{ file}$ , and  $P(l \text{ listing})$ , whenever  $l \text{ listing}$ , it is enough to show what?

**Solution:**

- $P(\text{empty listing})$
- If  $s \text{ bitstring}$ , then  $P(\text{ord}(s) \text{ file})$
- If  $P(d \text{ listing})$ , then  $P(\text{dir}(d) \text{ file})$
- If  $P(f \text{ file})$ , and  $P(d \text{ listing})$ , then  $P((f, d) \text{ listing})$

## Dynamic Scope

In a dynamically-scoped language, function application is defined to use replacement, instead of substitution. Instead of renaming variables in the function body to avoid capture, dynamic-scoping is defined to incur variable capture. For example, let us define  $e = \lambda x : \text{nat} . \lambda y : \text{nat} . x$ , and  $e' = y + 1$ . Evaluation of expressions containing free variables is defined, but evaluating a free variable is not. In this example,  $\cdot \vdash e : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$ , and  $y : \text{nat} \vdash e' : \text{nat}$ . The evaluation of  $e(e')$  in a dynamically-scoped language steps to  $\lambda y : \text{nat} . y + 1$ , where the free variable  $y$  is captured.

- (f) (5 points) Assume  $|x|$  represents the length of a string. That is,  $x : \text{str} \vdash |x| : \text{nat}$ . What goes wrong when the following is evaluated?  
 $(\lambda f : \text{str} \rightarrow \text{nat} . \lambda y : \text{str} . f(y))(\lambda x : \text{str} . y + |x|)$

**Solution:** This steps to  $\lambda y : \text{str} . ((\lambda x : \text{str} . y + |x|)(y))$ , which cannot be well-typed.  $y$  must be both a  $\text{str}$ , and a  $\text{nat}$

## Question 2 [35]: Type Safety

In this problem, we are going to explore a small object-oriented language.

### Syntax

We begin by giving the syntax of our language.

$$\begin{aligned} C, D &\in \text{CLASSES} \\ \tau &::= \text{Object}[C] \\ e &::= \text{new}[C] \mid \text{cast}[C](e) \end{aligned}$$

The classes of this language are elements of the finite set `CLASSES`, organized in a class hierarchy. The meta-variables  $C, D$  represent these classes. All rules have the implicit assumption that  $C, D \in \text{CLASSES}$  for all named variables  $C, D$ .  $\sqsubseteq$  is the subclass relation over types.  $C \sqsubseteq D$  means that  $C$  is a subclass of  $D$ .  $C \not\sqsubseteq D$  means that  $C$  is not a subclass of  $D$ .

The only type in this language is `Object[C]`. This states that the runtime class of this object will be tagged with some class  $D$ , such that  $D \sqsubseteq C$ .

### Static Semantics

$$\begin{array}{c} \frac{}{\Gamma \vdash \text{new}[C] : \text{Object}[C]} \text{ TNew} \quad \frac{\Gamma \vdash e : \text{Object}[C]}{\Gamma \vdash \text{cast}[D](e) : \text{Object}[D]} \text{ TCast} \\ \frac{\Gamma \vdash e : \text{Object}[C] \quad C \sqsubseteq D}{\Gamma \vdash e : \text{Object}[D]} \text{ TSubclass} \end{array}$$

The typing rule, *TNew*, states that constructing a new class  $C$  has type `Object[C]`. *TCast* allows you to perform a runtime cast to assert that the expression  $e$  is tagged with a subclass of  $D$ . The dynamic semantics, shown below, will allow the cast to succeed only if the runtime tag of the object,  $C$ , is a subclass of  $D$ . *TSubclass* lets you perform an implicit cast – if an expression can be typed `Object[C]`, then it can also be typed to a superclass, `Object[D]` (that is,  $C \sqsubseteq D$ ).

### Dynamic Semantics

The value judgement for this language is

$$\frac{}{\text{new}[C] \text{ value}} \text{ VNew}$$

The only value of our language is the expression `new[C]`.

**Lemma 1.** (*Canonical Forms for Classes*) If  $\cdot \vdash e : \text{Object}[D]$  and  $e$  value, then  $e = \text{new}[C]$ , where  $C \leq D$

$$\frac{e \mapsto e'}{\text{cast}[C](e) \mapsto \text{cast}[C](e')} \text{ECast}_1$$

$$\frac{C \leq D}{\text{cast}[D](\text{new}[C]) \mapsto \text{new}[C]} \text{ECast}_2$$

The only closed expression in our language is currently of the form  $\text{cast}[C_1](\text{cast}[C_2](\dots\text{cast}[C_k](\text{new}[C])))$ . However, this can be extended with many other interesting constructs (field access, method calls, etc...). Even this limited language represents the core concepts of an object-oriented design.

### Proving Preservation

In order to prove preservation, we will need to make use of the inversion lemma. The inversion lemma is a bit different than you are used to in the presence of the *TSubclass* rule.

**Lemma 2.** (*Inversion*) If  $\Gamma \vdash e : \tau$ , then

- if  $e = \text{new}[C]$ , then  $\tau = \text{Object}[D]$ , and  $C \leq D$
- if  $e = \text{cast}[C](e')$ , then  $\tau = \text{Object}[D]$ ,  $C \leq D$ , and  $\Gamma \vdash e' : \text{Object}[C']$

You do not need to prove this inversion lemma.

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

**Lemma 3.** (*Preservation*) If  $\cdot \vdash e : \tau$ , and  $e \mapsto e'$ , then  $\cdot \vdash e' : \tau$

Your task is to prove this theorem by rule induction on the derivation of  $e \mapsto e'$ . For each of the two cases, *ECast*<sub>1</sub>, and *ECast*<sub>2</sub>, explicitly state what you may assume, and what you need to prove.

(a) (7 points) case *ECast*<sub>1</sub>:

**Solution:** We may assume that (a)  $\text{cast}[C](e) \mapsto \text{cast}[C](e')$ , and (b)  $e \mapsto e'$ , and (c)  $\cdot \vdash \text{cast}[C](e) : \tau$ . We need to show that  $\cdot \vdash \text{cast}[C](e') : \tau$

1.  $C \leq D$ ,  $\tau = \text{Object}[D]$  by inversion of *TCast*, and c
2.  $\cdot \vdash e : \text{Object}[C']$ , by inversion of *TCast*, and c
3.  $\cdot \vdash e' : \text{Object}[C']$ , by IH on 2, b
4.  $\cdot \vdash \text{cast}[C](e') : \text{Object}[C]$ , by *TCast*, and 3
5.  $\cdot \vdash \text{cast}[C](e') : \text{Object}[D]$ , by *TSubclass*, 1, 4

(b) (7 points) case  $ECast_2$ :

**Solution:** We may assume that (a)  $\text{cast}[D](\text{new}[C]) \mapsto \text{new}[C]$ , and (b)  $C \leq D$ , and (c)  $\cdot \vdash \text{cast}[D](\text{new}[C]) : \tau$ . We need to show that  $\cdot \vdash \text{new}[C] : \tau$

1.  $D \leq E$ , and  $\tau = \text{Object}[E]$ , by inversion of  $TCast$ , and c
2.  $\cdot \vdash \text{new}[C] : \text{Object}[C]$ , by  $TNew$
3.  $\cdot \vdash \text{new}[C] : \text{Object}[D]$ , by  $TSubclass$ , b, 2
4.  $\cdot \vdash \text{new}[C] : \text{Object}[E]$ , by  $TSubclass$ , 3, 1
5.  $\cdot \vdash \text{new}[C] : \tau$ , by  $TSubclass$ , 1, 4

### Proving Progress

Unfortunately, our language is not type-safe. Specifically, the following progress theorem is not provable.

**Lemma 4.** (*ProgressUnsafe*) *If  $\cdot \vdash e : \tau$ , then either  $e$  value, or  $e \mapsto e'$ .*

(c) (3 points) Show a well-typed term for which the progress theorem does not hold. For this, you will need to not only define the term, but also the list of classes, and the subtyping relation between them.

**Solution:**  $\cdot \vdash \text{cast}[C](\text{new}[D]) : \text{Object}[C]$ , where  $D \not\leq C$

One way to fix the soundness of the language is a new `InvalidCast` expression to the language.

We will also add a new judgement,  $e \text{ error}$ .

$$\frac{}{\text{InvalidCast error}} \text{ErrInvalidCast}$$

(d) (4 points) Add any additional rules for the  $e \text{ error}$  judgement.

**Solution:**

$$\frac{e \text{ error}}{\text{cast}[C](e) \text{ error}} \text{ErrCast}$$

(e) (3 points) Add any additional rules for the  $e \mapsto e'$  judgement

**Solution:**

$$\frac{C \not\leq D}{\text{cast}[D](\text{new}[C]) \mapsto \text{InvalidCast}} \text{ECast}_3$$

(f) (4 points) Restate the progress theorem to allow for the possibility of errors

**Solution:** If  $\cdot \vdash e : \tau$ , then either  $e$  value, or  $e \mapsto e'$ , or  $e$  error.

- (g) (7 points) Allowing for errors, the progress theorem can now be proven by case induction on the derivation of  $\cdot \vdash e : \tau$ . Prove that progress holds for the cast  $TCast$

**Solution:** Assuming a)  $\cdot \vdash \text{cast}[D](e) : \text{Object}[D]$ , and b)  $\cdot \vdash e : \text{Object}[C]$ , we need to show that either  $\text{cast}[D](e)$  value, or  $\text{cast}[D](e) \mapsto e'$  for some  $e'$ , or  $\text{cast}[D](e)$  error

1. Either i)  $e$  value, or ii)  $e \mapsto e''$ , or iii)  $e$  error, by IH and b
2. Assuming 1.i
  - (a)  $e = \text{new}[C']$ , by 1.i, and canonical forms lemma
  - (b) Assuming  $C' \preceq D$ ,  $\text{cast}[D](e) \mapsto \text{new}[C']$ , by a,  $ECast_2$
  - (c) Assuming  $C' \not\preceq D$ ,  $\text{cast}[D](e) \mapsto \text{InvalidCast}$  by a,  $ECast_3$
3. Assuming 1.ii
  - (a)  $\text{cast}[D](e) \mapsto \text{cast}[D](e'')$ , by 1.ii and  $ECast_1$
4. Assuming 1.iii
  - (a)  $\text{cast}[D](e)$  error, by 1.iii and  $ErrCast$

### Question 3 [30]: Loopy Lists

In this question we consider a form of infinite list of natural numbers, called a *loopy list*, consisting of nodes of the form  $\text{cons}(e_1, x.e_2)$ , where  $e_1$  is the head element,  $x$  stands for the node itself, and  $e_2$  is the tail, another loopy list. We could also admit an empty list,  $\text{nil}$ , but for the sake of simplicity in this question we omit it. Consequently, every loopy list is “d-shaped” in that it consists of a sequence of elements that eventually loops back on itself.

For example, the loopylist

$$\text{cons}(\text{zero}, x.\text{cons}(\text{zero}, y.x))$$

looks like a circle, whereas the loopylist

$$\text{cons}(\text{zero}, x.\text{cons}(\text{zero}, y.y))$$

has a tail on it before it begins looping.

Loopylists have two elimination forms:  $\text{hd}(e)$  and  $\text{tl}(e)$  whose static semantics is as expected:

$$\frac{\Gamma \vdash e : \text{loopylist}}{\Gamma \vdash \text{hd}(e) : \text{nat}} \quad \frac{\Gamma \vdash e : \text{loopylist}}{\Gamma \vdash \text{tl}(e) : \text{loopylist}}$$

- (a) (6 points) Write the typing rule for the  $\text{cons}(e_1, x.e_2)$  constructor for loopy lists.

**Solution:**

$$\frac{\Gamma \vdash e_1 : \text{nat} \quad \Gamma, x : \text{loopylist} \vdash e_2 : \text{loopylist}}{\Gamma \vdash \text{cons}(e_1, x.e_2) : \text{loopylist}} \text{ TLLCons}$$

- (b) The dynamics of loopy lists includes the following familiar-looking rules:

$$\frac{e_1 \text{ value}}{\text{cons}(e_1, x.e_2) \text{ value}} \quad \frac{e_1 \mapsto e'_1}{\text{cons}(e_1, x.e_2) \mapsto \text{cons}(e'_1, x.e_2)}$$

$$\frac{e \mapsto e'}{\text{hd}(e) \mapsto \text{hd}(e')} \quad \frac{e \mapsto e'}{\text{tl}(e) \mapsto \text{tl}(e')}$$

$$\frac{e_1 \text{ value}}{\text{hd}(\text{cons}(e_1, x.e_2)) \mapsto e_1}$$

- i. (6 points) Give the missing rule for evaluating  $\text{tl}(e)$  in that case that  $e$  value.

**Solution:**

$$\frac{e_1 \text{ value}}{\text{tl}(\text{cons}(e_1, x.e_2)) \mapsto [\text{cons}(e_1, x.e_2)/x]e_2}$$

- (c) Previously, we have shown that lists are a specific instance of a more general case of recursive datatypes - specifically, the recursive type  $\mu t.\text{unit} + \text{nat} \times t$ . In addition, we introduced recursion using only recursive types and functions using the type  $\tau \text{ self}$ , represented as the recursive type  $\mu t.t \rightarrow \tau$  for any  $\tau$ . Intuitively, it seems like there should be some way to represent loopylists as a recursive type, too.

- i. (6 points) Give a definition of the type `loopylist` using recursive types. *Be sure to account for self-reference!*

**Solution:**

$$\text{loopylist} := \mu t. (t \rightarrow \text{nat} \times t)$$

- ii. (6 points) Define the constructor `cons(e1, x.e2)` in terms of the recursive type given in the preceding question. Your answer should make use of the `fold` operation for a recursive type.

**Solution:**

$$\text{cons}(e_1, x.e_2) := \text{fold}(\lambda(x : \tau. \langle e_1, e_2 \rangle))$$

where  $\tau$  is as above.

- iii. (6 points) Define the operators `hd(e)` and `tl(e)` in terms of the recursive type definition of `loopylist`. Your answer should make use of the `unfold` operation for a recursive type.

**Solution:**

$$\text{hd}(e) := \text{fst}(\text{unfold}(e) e)$$

$$\text{tl}(e) := \text{snd}(\text{unfold}(e) e)$$

**Scratch Work:**

**Scratch Work:**