

Midterm Examination

15-312: Foundation of Programming Languages

March, 2008

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 Answer [25 Points]

For yes/no questions, circle either **yes** or **no**. You don't need to give any explanation.

1.1 Gödel's T

The syntax of Gödel's T language is defined as follow:

$$\begin{aligned}\tau & ::= \text{nat} \mid \text{arrow}(\tau, \tau') \\ e & ::= \text{zero} \mid \text{succ}(e) \mid \text{rec}[\tau](e_1, e_2, x.y.e_3) \mid \\ & \quad x \mid \text{app}(e_1, e_2) \mid \text{lambda}[\tau](x.e_1)\end{aligned}$$

Task 1.1 (5%). Are these two terms α -equivalent: **YES** **NO**

$\text{lambda}[\tau](x.\text{rec}[\tau](x, \text{zero}, x.y.x))$

$\text{lambda}[\tau](y.\text{rec}[\tau](y, \text{zero}, y.x.y))$

Task 1.2 (5%). Which mathematical function does this Gödel expression correspond to?

$\text{lambda}[\text{nat}](a.\text{lambda}[\text{nat}](b.\text{rec}[\text{nat}](a, b, x.y.\text{succ}(\text{succ}(y))))))$

Your answer:

1.2 Plotkin's PCF

The syntax of Plotkin's PCF is defined as follow:

$$\begin{aligned}\tau & ::= \text{nat} \mid \text{arrow}(\tau, \tau') \\ e & ::= \text{zero} \mid \text{succ}(e) \mid \text{ifz}(e, e_0, x.e_1) \mid \\ & \quad x \mid \text{app}(e_1, e_2) \mid \text{lambda}[\tau](x.e_1) \mid \\ & \quad \text{fix}[\tau](x.e)\end{aligned}$$

Task 1.3 (5%). Does this PCF function terminate on all inputs? **YES** **NO**

$\text{fix}[\text{arrow}(\text{nat}, \text{nat})](f.\text{lambda}[\text{nat}](x.\text{ifz}(x, \text{zero}, y.\text{app}(f, \text{succ}(x))))))$

Task 1.4 (5%). Which mathematical function does this PCF expression correspond to?

```
fix[arrow(nat, nat)](f.lambda[nat](x.ifz(x, succ(zero), x'.app(app(times, x), app(f, x')))))
```

where we can assume that $times : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$ is a correct implementation of multiplication over nat 's.

Your answer:

1.3 Recursive Types

Task 1.5 (5%). In a language with labeled sums, products and recursive types, give a recursive type corresponding to the following ML datatype:

```
datatype stack = Empty | Push of stack * int
```

Your answer:

2 Rule Induction [25 Points]

Parentheses, brackets, matching tags, etc., have the purpose to grouping expressions in a language so that they are handled as a unit by surrounding syntax. These *delimiters* work in pairs, with an opening token (e.g., “[”, “\begin{itemize}”, and “<P>”) and a corresponding closing token (respectively, ”]”, “\end{itemize}”, and “</P>”). In order to achieve this purpose, they should be *balanced* in a text so that for every opening token there is a corresponding closing token of the same kind, and vice versa, and that the text in between is itself balanced. Good editors provide a mechanism to check that all delimiters are balanced.

Let us formalize this notion. We will simplify the problem by considering strings consisting only of delimiters, and write [for an opening token, and] for the corresponding closing token. The notion of a string s being balanced is expressed through the judgment “ s balanced”, defined by the following rules:

$$\frac{}{\epsilon \text{ balanced}} \text{ } b\epsilon \quad \frac{s \text{ balanced}}{[\cdot s \cdot] \text{ balanced}} \text{ } obc \quad \frac{s_1 \text{ balanced} \quad s_2 \text{ balanced}}{s_1 \cdot s_2 \text{ balanced}} \text{ } bb$$

We rely on the usual definition of strings: ϵ is the empty string, $_{-} \cdot _{-}$ denotes string concatenation, and it is implicitly assumed that concatenation is associative (so that $(s_1 \cdot s_2) \cdot s_3 = s_1 \cdot (s_2 \cdot s_3)$). Examples of balanced and unbalanced strings of delimiters follow (in concrete syntax, for readability):

<i>Balanced</i>	<i>Unbalanced</i>
[] [[] []]] [
[[[]]]	[[[]
[] [] [] []]]]

This definition does not lend itself to an efficient mechanization. If we have a way to remember open delimiters in a stack as we encounter them, we can design a simple push-down automaton that decides whether a string is balanced, as each symbol is read, left to right. This is modeled by means of a *state* of the form “ $O \triangleright s$ ” where O contains the delimiters that have been Opened but not yet closed, and s is the part of the input string to be examined. The behavior of the automaton is given by the following two transition rules between states:

$$\frac{}{O \triangleright [\cdot s \mapsto O \cdot [\triangleright s} \triangleright^o \quad \frac{}{O \cdot [\triangleright] \cdot s \mapsto O \triangleright s} \triangleright^c$$

We want to prove that this automaton implements the above notion of a balanced string, more precisely, we want to show that

Proposition 2.1. *For any string s , if s balanced, then $\epsilon \triangleright s \mapsto^* \epsilon \triangleright \epsilon$.*

where “ $_ \mapsto^* _$ ” is the finite iteration of the step judgment “ $_ \mapsto _$ ”.

A direct proof by induction fails because we cannot apply the induction hypothesis once we have put something in the stack. We need to strengthen the statement:

Property 2.2.

For any string s , if s balanced, then $O \triangleright s \cdot s' \mapsto^ O \triangleright s'$ for any strings O and s' .*

We want you to prove proposition 2.2. The proof is by induction on a derivation of the judgment s balanced.

Task 2.1 (10%). Complete the partial proof of property 2.2 for the following case:

Case $\frac{}{\epsilon \text{ balanced}}_{be}$, with $s = \epsilon$:

TO SHOW: $O \triangleright \epsilon \cdot s' \mapsto^* O \triangleright s'$ for any strings O and s' .

Proof:

Task 2.2 (15%). Complete the partial proof of property 2.2 for the following case:

Case $\frac{s_1 \text{ balanced}}{[\cdot s_1 \cdot] \text{ balanced}}_{obc}$, with $s = [\cdot s_1 \cdot]$:

TO SHOW: $O \triangleright [\cdot s_1 \cdot] \cdot s' \mapsto^* O \triangleright s'$ for any strings O and s' .

By the IH:

Proof:

Bonus Task Complete the partial proof of property 2.2 for the following case:

Case $\frac{s_1 \text{ balanced } s_2 \text{ balanced}}{s_1 \cdot s_2 \text{ balanced}}$ bb , with $s = s_1 \cdot s_2$:

TO SHOW:

By the IH:

Proof:

3 Type Safety [25 Points]

Let us consider a typed λ -calculus with lists:

$$\begin{aligned} \tau &::= \text{arrow}(\tau, \tau') \mid \text{list}(\tau) \\ e &::= x \mid \text{app}(e_1, e_2) \mid \text{lambda}[\tau](x.e_1) \mid \\ &\quad \text{nil} \mid \text{cons}(e_1, e_2) \mid \text{listcase}(e_1, e_2, x.y.e_3) \mid \\ &\quad \text{fix}[\tau](x.e) \end{aligned}$$

Static Semantics

$$\begin{array}{c} \overline{\Gamma, x : \tau \vdash x : \tau} \text{ of-var} \\ \\ \frac{\overline{\Gamma, x : \tau \vdash e : \tau'}}{\Gamma \vdash \text{lambda}[\tau](x.e) : \text{arrow}(\tau, \tau')} \text{ of-lambda} \quad \frac{\Gamma \vdash e_1 : \text{arrow}(\tau', \tau) \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash \text{app}(e_1, e_2) : \tau} \text{ app} \\ \\ \overline{\Gamma \vdash \text{nil} : \text{list}(\tau)} \text{ of-nil} \quad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash l : \text{list}(\tau)}{\Gamma \vdash \text{cons}(e, l) : \text{list}(\tau)} \text{ of-cons} \\ \\ \frac{\Gamma \vdash e_1 : \text{list}(\tau) \quad \Gamma \vdash e_2 : \tau' \quad \Gamma, x : \tau, y : \text{list}(\tau) \vdash e_3 : \tau'}{\Gamma \vdash \text{listcase}(e_1, e_2, x.y.e_3) : \tau'} \text{ of-case} \\ \\ \frac{\Gamma, x : \tau \vdash \text{fix}[\tau](x.e) : \tau}{\Gamma \vdash \text{fix}[\tau](x.e) : \tau} \text{ of-fix} \end{array}$$

Dynamic Semantics

$$\begin{array}{c} \overline{\text{lambda}[\tau](x.e) \text{ val}} \text{ lambda} \quad \overline{\text{nil val}} \text{ nil} \quad \overline{\text{cons}(e_1, e_2) \text{ val}} \text{ cons} \\ \\ \frac{e_1 \mapsto e'_1}{\text{app}(e_1, e_2) \mapsto \text{app}(e'_1, e_2)} \text{ ap1} \quad \overline{\text{app}(\text{lambda}[\tau](x.e_1), e_2) \mapsto [e_2/x]e_1} \text{ ap2} \\ \\ \frac{e_1 \mapsto e'_1}{\text{listcase}(e_1, e_2, x.y.e_3) \mapsto \text{listcase}(e'_1, e_2, x.y.e_3)} \text{ case}_1 \\ \\ \overline{\text{listcase}(\text{nil}, e_2, x.y.e_3) \mapsto e_2} \text{ case}_2 \quad \overline{\text{listcase}(\text{cons}(e, e_1), e_2, x.y.e_3) \mapsto [e/x, e_1/y]e_3} \text{ case}_3 \\ \\ \overline{\text{fix}[\tau](x.e) \mapsto [\text{fix}[\tau](x.e)/x]e} \text{ fix} \end{array}$$

Task 3.1 (10%). Assuming (without proving it) the *Substitution lemma* and the *inversion lemma* below, prove the *Type Preservation theorem* for the rule case_1 .

Theorem (Type Preservation) If $\cdot \vdash e : \tau$ and $e \mapsto e'$, then $\cdot \vdash e' : \tau$

Do not prove — Do not prove — Do not prove — Do not prove — Do not prove — Do not prove
Lemma (Substitution) If $\Gamma, x : \tau \vdash e' : \tau'$ and $\Gamma \vdash e : \tau$, then $\Gamma \vdash [e/x]e' : \tau'$
Lemma (Inversion for of-case) If $\Gamma \vdash \text{listcase}(e_1, e_2, x.y.e_3) : \tau'$,
 then $\Gamma \vdash e_1 : \text{list}(\tau)$ and $\Gamma \vdash e_2 : \tau'$ and $\Gamma, x : \tau, y : \text{list}(\tau) \vdash e_3 : \tau'$
Do not prove — Do not prove — Do not prove — Do not prove — Do not prove — Do not prove

The proof is by induction on $e \mapsto e'$.

Show only the following case:

- Case for case_1 :

TO SHOW: For all τ , if $\Gamma \vdash \text{listcase}(e_1, e_2, x.y.e_3) : \tau$ and $e_1 \mapsto e'_1$ then $\Gamma \vdash \text{listcase}(e'_1, e_2, x.y.e_3) : \tau$.

By the IH:

Proof:

Task 3.2 (15%). Assuming (without proving it) the *Canonical Forms lemma*, prove the *Progress theorem* for the rule `of-case`.

Theorem (Progress) If $\cdot \vdash e : \tau$, then either e val or there exists e' such that $e \mapsto e'$

$\left(\begin{array}{l} \textit{Do not prove} - \textit{Do not prove} - \textit{Do not prove} - \textit{Do not prove} - \textit{Do not prove} - \textit{Do not prove} \\ \mathbf{Lemma} \text{ (Canonical Forms)} \\ \bullet \text{ If } e \text{ val and } \Gamma \vdash e : \textit{list}(\tau), \text{ then either } e = \textit{nil} \text{ or } e = \textit{cons}(e_1, e_2) \text{ for some } e_1 \\ \text{ and } e_2 \text{ such that } \Gamma \vdash e_1 : \tau \text{ and } \Gamma \vdash e_2 : \textit{list}(\tau) \\ \textit{Do not prove} - \textit{Do not prove} - \textit{Do not prove} - \textit{Do not prove} - \textit{Do not prove} - \textit{Do not prove} \end{array} \right)$

The proof is by induction on $\cdot \vdash e : \tau$.

Show only the following case:

- Case for `of-case`:

TO SHOW:

By the IH:

Proof:

4 Lazy evaluation of streams [25 Points]

A stream is an infinite sequence of data each with the same type. We will model them with the following language, where natural numbers are used as the convenient base case.

$$\begin{aligned}
 \tau &::= \text{nat} \mid \text{arrow}(\tau, \tau') \mid \text{stream}(\tau) \\
 e &::= \text{zero} \mid \text{succ}(z) \\
 &\quad \mid x \mid \text{lambda}[\tau](x.e) \mid \text{app}(e_1, e_2) \\
 &\quad \mid \text{fix}[\tau](e) \\
 &\quad \mid \langle\langle e_1, e_2 \rangle\rangle \mid \text{head}(e) \mid \text{tail}(e)
 \end{aligned}$$

The language includes the following constructors:

- $\langle\langle e_1, e_2 \rangle\rangle$ has type $\text{stream}(\tau)$, it is a stream of elements of type τ . The term e_1 is the head of the stream containing the first data item, e_2 is the rest (or tail) of the stream.
- $\text{head}(e)$ is the head element of the stream e .
- $\text{tail}(e)$ is the tail stream of the stream e .
- As usual zero and $\text{succ}(e)$ are the constructors for natural numbers, $\text{lambda}[\tau](x.e)$ and $\text{app}(e_1, e_2)$ are abstraction and application for functions, and $\text{fix}[\tau](x.e)$ is the fix point operator.

Using this machinery, here are two expressions, one for a stream of zeros, *ZeroStream*, which stands for an infinite sequence of zero's, the other for the stream containing all the natural numbers, *AllNats*, which stands for the infinite sequence $(0, 1, 2, 3, \dots)$. In order to define *AllNats*, we first define the function *NatsFrom*(x) which will return the stream of all the natural numbers starting from x .

$$\begin{aligned}
 \text{ZeroStream} & \text{fix}[\text{stream}(\text{nat})](s.\langle\langle \text{zero}, s \rangle\rangle) \\
 \text{NatsFrom} & \text{fix}[\text{arrow}(\text{nat}, \text{stream}(\text{nat}))](nf.\text{lambda}[\text{nat}](x.\langle\langle x, \text{app}(nf, \text{succ}(x)) \rangle\rangle)) \\
 \text{AllNats} & \text{app}(\text{NatsFrom}, \text{zero})
 \end{aligned}$$

The operations, $\text{head}(-)$ and $\text{tail}(-)$, that access a stream must be carefully designed to ensure that the computation terminates. Consider for example the expression $\text{head}(\text{ZeroStream})$: if the language were eager, it would try to evaluate *ZeroStream* in its entirety before returning its first element, but this would never terminate because this stream is infinite. Instead the language must be *lazy*: the dynamic semantics should do just enough computation to permit the next operation to be executed. In this example, the evaluation of $\text{head}(\text{ZeroStream})$ should unroll *ZeroStream* just enough to expose its first element, no more. Similarly for $\text{tail}(\text{ZeroStream})$: it does not need to unroll the definition more than once.

The static and dynamic semantics of our language is partially defined as follows:

Static Semantics

$$\begin{array}{c}
 \frac{}{\Gamma, x : \tau \vdash x : \tau} \text{of-var} \quad \frac{}{\Gamma \vdash \text{zero} : \text{nat}} \text{of-z} \quad \frac{\Gamma \vdash n : \text{nat}}{\Gamma \vdash \text{succ}(n) : \text{nat}} \text{of-succ} \\
 \\
 \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \text{lambda}[\tau](x.e) : \text{arrow}(\tau, \tau')} \text{of-lambda} \quad \frac{\Gamma \vdash e_1 : \text{arrow}(\tau', \tau) \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash \text{app}(e_1, e_2) : \tau} \text{app} \\
 \\
 \frac{\Gamma, x : \tau \vdash \text{fix}[\tau](x.e) : \tau}{\Gamma \vdash \text{fix}[\tau](x.e) : \tau} \text{of-fix} \\
 \\
 \dots \text{ to be completed.}
 \end{array}$$

Dynamic Semantics

$$\begin{array}{c}
 \frac{}{\text{zero val}} \text{z} \quad \frac{e \text{ val}}{\text{succ}(e) \text{ val}} \text{succ} \quad \frac{}{\text{lambda}[\tau](x.e) \text{ val}} \text{lambda} \\
 \\
 \frac{e \mapsto e'}{\text{succ}(e) \mapsto \text{succ}(e')} \text{succ} \\
 \\
 \frac{e_1 \mapsto e'_1}{\text{app}(e_1, e_2) \mapsto \text{app}(e'_1, e_2)} \text{ap1} \quad \frac{}{\text{app}(\text{lambda}[\tau](x.e_1), e_2) \mapsto [e_2/x]e_1} \text{ap2} \\
 \\
 \frac{}{\text{fix}[\tau](x.e) \mapsto [\text{fix}[\tau](x.e)/x]e} \text{fix} \\
 \\
 \dots \text{ to be completed.}
 \end{array}$$

Task 4.1 (10%). According to the informal specification given above, complete the static semantics by giving the appropriate rules for $\langle\langle -, - \rangle\rangle$, $\text{tail}(-)$ and $\text{head}(-)$.

Static Semantics:

Task 4.2 (10%). According to the informal specification given above, complete the dynamic semantics by giving the appropriate rules allowing $\langle\langle -, - \rangle\rangle$, $\text{tail}(-)$ and $\text{head}(-)$ to be evaluated.

Dynamic Semantics:

Task 4.3 (5%). Using the language, write an expression of type $(\text{nat} \rightarrow \text{nat}) \rightarrow \text{stream}(\text{nat}) \rightarrow \text{stream}(\text{nat})$ that implements the map function for streams of type nat in either abstract or concrete syntax.

Your answer: