Midterm Exam

15-814 Types and Programming Languages
Frank Pfenning

October 9, 2025

Name: Andrew ID:

Instructions

- This exam is closed-book, closed-notes.
- You have 80 minutes to complete the exam.
- There are 4 problems.
- For reference, on pages 10–14 there is an appendix with the syntax, statics, and dynamics for our call-by-value language. You may tear away these pages and you do not need to hand them in.
- For code-related answers, you may use the abstract syntax we have mostly used in lecture or the concrete syntax of LAMBDA. We will not be particular about issues of syntax.

	λ-Calculus	Polymorphism	Nondeterminism	Type Isomorphisms	
	Prob 1	Prob 2	Prob 3	Prob 4	Total
Score					
Max	40	50	40	30	150+10

1 λ-Calculus (40 points)

The *untyped call-by-value λ-calculus* does not reduce all the way to a normal form, but simply uses the rules for values and stepping for functions and applications from our call-by-value language (see the appendix).

The *untyped call-by-name λ-calculus* uses the same notion of value, but does not reduce the arguments to functions.

For both of these (and the rest of this problem) we assume that we only reduce closed terms.

Task 1 (5 pts) Give the stepping rules for the untyped call-by-name λ-calculus.

Task 2 (15 pts) Prove that every closed expression in the call-by-name λ-calculus either can step or is a value. If it is an induction, make clear what the induction is over and show all relevant cases in the proof.

The combinators Y and L are defined by

$$\begin{array}{rcl} \mathsf{Y} & = & \lambda f.\,(\lambda x.\, f\,(x\,x))\,(\lambda x.\, f\,(x\,x)) \\ \mathsf{L} & = & \lambda u.\,\lambda w.\, w \end{array}$$

Task 3 (10 pts) Show each step in the computation of $\mathsf{Y}\ \mathsf{L}$ to a value in the call-by-name λ-calculus or indicate the computation does not terminate. Note that replacing combinators by their definition does not count as a step since their definitions are made at the metalevel.

Task 4 (10 pts) Show each step in the computation of $\mathsf{Y}\ \mathsf{L}$ to a value in the call-by-value λ-calculus or indicate the computation does not terminate.

2 Polymorphic Encoding of Data (50 points)

Consider binary trees (type tree) with constructors leaf : tree and node : tree $\to$ tree $\to$ tree. Note that trees in this form do not contain any data.

Task 5 (5 pts) Write out the *schema of iteration* over binary trees.

Task 6 (10 pts) Give a type of the encoding of binary trees as their iterator in the polymorphic λ-calculus. Call this type tree.

Task 7 (10 pts) Define the constructors leaf : tree and node : tree $\to$ tree $\to$ tree.

Task 8 (5 pts) Give an encoding of trees in our call-by-value language. Call this type wood.

The polymorphic encoding of trees works even in our call-by-value language, but we cannot directly observe the outcome of computations because values of the polymorphic tree type are not observable.

Task 9 (10 pts) Define a function observe : tree $\rightarrow$ wood that allows us to observe the structure of a tree.

Task 10 (10 pts) Define a function reflect : wood $\rightarrow$ tree that converts an observable tree to its polymorphic representation such that observe and reflect are witnesses to an isomorphism between the two types. You don't need to show this property.

3 Nondeterminism (40 points)

Consider an extension to our call-by-value language (see appendix) by an operator of *nondeterministic choice*: $e_1 \oplus e_2$ should be able to step to e_1 and e_2. For example, with

$\mathsf{nat} = \mu\mathsf{nat}.\ (\mathbf{z} : 1) + (\mathbf{s} : \mathsf{nat})$
$\mathsf{zero} = \mathsf{fold}\ (\mathbf{z} \cdot (\,))$
$\mathsf{succ} = \lambda n.\ \mathsf{fold}\ (\mathbf{s} \cdot n)$

$\mathsf{one/two} = \mathsf{succ}\ (\mathsf{zero} \oplus (\mathsf{succ\ zero}))$

we should have that one/two evaluates to (the representations of) 1 and 2.

$\mathsf{one/two} \mapsto^* \ulcorner 1 \urcorner$ and $\mathsf{one/two} \mapsto^* \ulcorner 2 \urcorner$

You should design the rules so that preservation and progress continue to be hold (to the extent that this is possible), while clearly we no longer have sequentiality (sometimes call small-step determinism).

Task 11 (5 pts) Give the new rule(s) for the judgments e *value* and $e \mapsto e'$ as necessary.

Task 12 (5 pts) Give the new rule(s) for typing $e_1 \oplus e_2$.

Now we add to our language the expression **fail** which represents a *nullary nondeterministic choice* which is intuitively the same as *failure*. That is **fail** does not step, but it is also not a value. It should be a consequence of your rules that, for example, $(\mathsf{succ}\ \mathbf{fail}) \oplus e \mapsto^* v$ if and only if $e \mapsto^* v$.

Task 13 (5 pts) Give the new typing rule(s) for typing **fail**.

For the purpose of the progress theorem, we now define a judgment $e \not\text{\textit{z}}$ which is true if e does not step *due explicit failure*. For example, $(\,)\ (\lambda x.\, x)$ is ill-typed and does not step, but we should **not** have $(\,)\ (\lambda x.\, x) \not\text{\textit{z}}$. On the other hand $\mathbf{fail}\ (\,) \not\text{\textit{z}}$ should hold.

Task 14 (15 pts) Give the rules for the $e \not\text{\textit{z}}$ judgment for the fragment of our language with functions and unit (including the case expression for unit), nondeterministic choice, and **fail**.

Task 15 (10 pts) Formulate a new version of the progress theorem that holds in the presence of nondeterministic choice and explicit failure. You do not need to prove it. Are the cases in the statement of the theorem mutually exclusive?

4 Isomorphisms and Retracts (30 points)

We defined that τ is *isomorphic* to σ if there are two functions (expressible in our language) $\mathsf{forth} : \tau \to \sigma$ and $\mathsf{back} : \sigma \to \tau$ such that both $\mathsf{back} \circ \mathsf{forth}$ and $\mathsf{forth} \circ \mathsf{back}$ are extensionally equal to the identity.

In this problem we restrict ourselves to the fragment of our language without recursive types and fixed points.

Task 16 (5 pts) Show that there is a τ such that τ **is not** isomorphic to $\tau \times \tau$.

Task 17 (5 pts) Show that there is a τ such that τ **is** isomorphic to $\tau \times \tau$.

We say that τ is a *retract* of σ if there are two functions (expressible in our language) forth : $\tau \to \sigma$ and back : $\sigma \to \tau$ such that back $\circ$ forth is extensionally equal to the identity on τ. This is "half" of the requirements for an isomorphism.

Task 18 (10 pts) Is τ a retract of $\tau \times \tau$ for all types τ? If yes, exhibit the function forth and back and show that they compose to the identity in the required direction. If not, give a counterexample in terms of a concrete τ.

Task 19 (10 pts) Is τ a retract of $\tau + \sigma$ for all types τ and σ? If yes, exhibit the functions forth and back and show that they compose to the identity in the required direction. If not, give a counterexample in terms of concrete τ and σ.

Appendix: Language Reference

Abstract Syntax

Types	τ	$::=$	$\tau_1 \to \tau_2 \mid \forall\alpha.\,\tau \mid \alpha \mid \&_{i\in I}(i : \tau_i)$	
		$\mid$	$\tau_1 \times \tau_2 \mid 1 \mid \sum_{i\in I}(i : \tau_i) \mid \mu\alpha.\,\tau$	
Terms	e	$::=$	$x \mid \lambda x.\,e \mid e_1\,e_2$	$(\to)$
		$\mid$	$\Lambda\alpha.\,e \mid e\,[\tau]$	$(\forall)$
		$\mid$	$\langle\!\vert i \Rightarrow e_i \vert\!\rangle_{i\in I} \mid e.i$	$(\&)$
		$\mid$	$\langle e_1, e_2\rangle \mid \mathsf{case}\ e\ (\langle x_1, x_2\rangle \Rightarrow e')$	$(\times)$
		$\mid$	$\langle\,\rangle \mid \mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e')$	(1)
		$\mid$	$k \cdot e \mid \mathsf{case}\ e\ (i \cdot x_i \Rightarrow e_i)_{i\in I}$	$(+)$
		$\mid$	$\mathsf{case}\ e\ (\,)$	(0)
		$\mid$	$\mathsf{fold}\ e \mid \mathsf{unfold}\ e$	(μ)
		$\mid$	$\mathsf{fix}\,f.\,e \mid f$	
Contexts	Γ	$::=$	$\cdot \mid \Gamma, \alpha\ \mathit{type} \mid \Gamma, x : \tau$	(all variables distinct)

Judgments

$\Gamma\ \mathit{ctx}$	Γ is a valid context	
$\Gamma \vdash \tau\ \mathit{type}$	τ is a valid type	presupposes $\Gamma\ \mathit{ctx}$
$\Gamma \vdash e : \tau$	expression e has type τ	presupposes $\Gamma\ \mathit{ctx}$, ensures $\Gamma \vdash \tau\ \mathit{type}$
$e\ \mathit{value}$	expression e is a value	presupposes $\cdot \vdash e : \tau$ for some τ
$e \mapsto e'$	expression e steps to e'	presupposes $\cdot \vdash e : \tau$ for some τ

Theorems

Preservation. If $\cdot \vdash e : \tau$ and $e \mapsto e'$ then $\cdot \vdash e' : \tau$.

Progress. For every expression $\cdot \vdash e : \tau$ either $e \mapsto e'$ for some e' or e *value*.

Finality of Values. If $\cdot \vdash e : \tau$ and e *value* then there is no e' with $e \mapsto e'$.

Sequentiality. If $\cdot \vdash e : \tau$ and $e \mapsto e_1$ and $e \mapsto e_2$ then $e_1 = e_2$.

Canonical Forms. Assume $\cdot \vdash e : \tau$ and e *value*.

(i) If $\tau = \tau_1 \to \tau_2$ then $e = \lambda x.\,e_2$ for some e_2

(ii) If $\tau = \forall\alpha.\,\tau'$ then $e = \Lambda\alpha.\,e'$ for some e'

(iii) If $\tau = \&_{i\in I}(i : \tau_i)$ then $e = \langle\!\vert i \Rightarrow e_i \vert\!\rangle_{i\in I}$ for some e_i

(iv) If $\tau = \tau_1 \times \tau_2$ then $e = \langle e_1, e_2\rangle$ for some e_1 *value* and e_2 *value*

(v) If $\tau = 1$ then $e = \langle\,\rangle$

(vi) If $\tau = \sum_{i\in I}(i : \tau_i)$ then $e = k \cdot e_k$ for some $k \in I$ and e_k *value*.

(vii) If $\tau = \mu\alpha.\,\tau'$ then $e = \mathsf{fold}\ e'$ for some e' *value*

Contexts Γ

$$\frac{}{(\cdot)\ \mathit{ctx}}\ \textsf{ctx/emp} \qquad \frac{\Gamma\ \mathit{ctx}}{(\Gamma, \alpha\ \mathit{type})\ \mathit{ctx}}\ \textsf{ctx/tpvar} \qquad \frac{\Gamma\ \mathit{ctx} \quad \Gamma \vdash \tau\ \mathit{type}}{(\Gamma, x : \tau)\ \mathit{ctx}}\ \textsf{ctx/var}$$

Functions $\tau_1 \to \tau_2$

$$\frac{\Gamma \vdash \tau_1\ \mathit{type} \quad \Gamma \vdash \tau_2\ \mathit{type}}{\Gamma \vdash \tau_1 \to \tau_2\ \mathit{type}}\ \textsf{tp/arrow}$$

$$\frac{\Gamma \vdash \tau_1\ \mathit{type} \quad \Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \lambda x_1{:}\tau_1.\, e_2 : \tau_1 \to \tau_2}\ \textsf{tp/lam} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsf{tp/var} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}\ \textsf{tp/app}$$

$$\frac{}{\lambda x.\, e\ \mathit{value}}\ \textsf{val/lam} \qquad \frac{e_2\ \mathit{value}}{(\lambda x.\, e_1)\, e_2 \mapsto [e_2/x]e_1}\ \textsf{step/app/lam}$$

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2}\ \textsf{step/app}_1 \qquad \frac{e_1\ \mathit{value} \quad e_2 \mapsto e_2'}{e_1\, e_2 \mapsto e_1\, e_2'}\ \textsf{step/app}_2$$

Polymorphic Types $\forall\alpha.\, \tau$

$$\frac{\alpha\ \mathit{type} \in \Gamma}{\Gamma \vdash \alpha\ \mathit{type}}\ \textsf{tp/tpvar} \qquad \frac{\Gamma, \alpha\ \mathit{type} \vdash \tau\ \mathit{type}}{\Gamma \vdash \forall\alpha.\, \tau\ \mathit{type}}\ \textsf{tp/forall}$$

$$\frac{\Gamma, \alpha\ \mathit{type} \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.\, e : \forall\alpha.\, \tau}\ \textsf{tp/tplam} \qquad \frac{\Gamma \vdash e : \forall\alpha.\, \tau \quad \Gamma \vdash \sigma\ \mathit{type}}{\Gamma \vdash e\, [\sigma] : [\sigma/\alpha]\tau}\ \textsf{tp/tpapp}$$

$$\frac{}{\Lambda\alpha.\, e\ \mathit{value}}\ \textsf{val/tplam} \qquad \frac{}{(\Lambda\alpha.\, e)\, [\tau] \mapsto [\tau/\alpha]e}\ \textsf{step/tpapp/tplam} \qquad \frac{e \mapsto e'}{e\, [\tau] \mapsto e'\, [\tau]}\ \textsf{step/tpapp}$$

Lazy Records $\&_{i\in I}(i : \tau_i)$

$$\frac{\Gamma \vdash \tau_i \; type \quad (\text{for all } i \in I)}{\Gamma \vdash \&_{i\in I}(\tau_i) \; type} \; \textsf{tp/lrecord}$$

$$\frac{\Gamma \vdash e_i : \tau_i \quad (\text{for all } i \in I)}{\Gamma \vdash \langle\!| i \Rightarrow e_i |\!\rangle_{i\in I} : \&_{i\in I}(i : \tau_i)} \; \textsf{tp/record} \qquad \frac{(k \in I) \quad \Gamma \vdash e : \&_{i\in I}(i : \tau_i)}{\Gamma \vdash e.k : \tau_k} \; \textsf{tp/proj}$$

$$\frac{}{\langle\!| i \Rightarrow e_i |\!\rangle_{i\in I} \; value} \; \textsf{val/record} \qquad \frac{}{\langle\!| i \Rightarrow e_i |\!\rangle_{i\in I}.k \mapsto e_k} \; \textsf{step/record/proj}$$

$$\frac{e \mapsto e'}{e.k \mapsto e'.k} \; \textsf{step/record}_0$$

Pairs $\tau_1 \times \tau_2$

$$\frac{\Gamma \vdash \tau_1 \; type \quad \Gamma \vdash \tau_2 \; type}{\Gamma \vdash \tau_1 \times \tau_2 \; type} \; \textsf{tp/prod}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \; \textsf{tp/pair} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e' : \tau'}{\Gamma \vdash \textsf{case } e \; (\langle x_1, x_2 \rangle \Rightarrow e') : \tau'} \; \textsf{tp/casep}$$

$$\frac{e_1 \; value \quad e_2 \; value}{\langle e_1, e_2 \rangle \; value} \; \textsf{val/pair} \qquad \frac{v_1 \; value \quad v_2 \; value}{\textsf{case } \langle v_1, v_2 \rangle \; (\langle x_1, x_2 \rangle \Rightarrow e_3) \mapsto [v_1/x_1][v_2/x_2]e_3} \; \textsf{step/casep/pair}$$

$$\frac{e_1 \mapsto e_1'}{\langle e_1, e_2 \rangle \mapsto \langle e_1', e_2 \rangle} \; \textsf{step/pair}_1 \qquad \frac{v_1 \; value \quad e_2 \mapsto e_2'}{\langle v_1, e_2 \rangle \mapsto \langle v_1, e_2' \rangle} \; \textsf{step/pair}_2$$

$$\frac{e_0 \mapsto e_0'}{\textsf{case } e_0 \; (\langle x_1, x_2 \rangle \Rightarrow e_3) \mapsto \textsf{case } e_0' \; (\langle x_1, x_2 \rangle \Rightarrow e_3)} \; \textsf{step/casep}_0$$

Unit 1

$$\frac{}{\Gamma \vdash 1\ \mathit{type}}\ \textsf{tp/one} \qquad \frac{}{\Gamma \vdash \langle\,\rangle : 1}\ \textsf{tp/unit} \qquad \frac{\Gamma \vdash e : 1 \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash \textsf{case}\ e\ (\langle\,\rangle \Rightarrow e') : \tau'}\ \textsf{tp/caseu}$$

$$\frac{}{\langle\,\rangle\ \mathit{value}}\ \textsf{val/unit} \qquad \frac{}{\textsf{case}\ \langle\,\rangle\ (\langle\,\rangle \Rightarrow e) \mapsto e}\ \textsf{step/caseu/unit}$$

$$\frac{e_0 \mapsto e_0'}{\textsf{case}\ e_0\ (\langle\,\rangle \Rightarrow e_1) \mapsto \textsf{case}\ e_0'\ (\langle\,\rangle \Rightarrow e_1)}\ \textsf{step/caseu}_0$$

Sums $\sum_{i \in I}(i : \tau_i)$

$$\frac{\Gamma \vdash \tau_i\ \mathit{type} \quad (\text{for all } i \in I)}{\Gamma \vdash \sum_{i \in I}(i : \tau_i)\ \mathit{type}}\ \textsf{tp/sum}$$

$$\frac{(k \in I) \quad \Gamma \vdash e_k : \tau_k \quad \Gamma \vdash \sum_{i \in I}(i : \tau_i)\ \mathit{type}}{\Gamma \vdash k \cdot e_k : \sum_{i \in I}(i : \tau_i)}\ \textsf{tp/tag}$$

$$\frac{\Gamma \vdash e : \sum_{i \in I}(i : \tau_i) \quad \Gamma, x_i : \tau_i \vdash e_i : \sigma \quad (\text{for all } i \in I)}{\Gamma \vdash \textsf{case}\ e\ (i \cdot x_i \Rightarrow e_i)_{i \in I} : \sigma}\ \textsf{tp/cases}$$

$$\frac{e\ \mathit{value}}{k \cdot e\ \mathit{value}}\ \textsf{val/tag} \qquad \frac{v_k\ \mathit{value}}{\textsf{case}\ k \cdot v_k\ (i \cdot x_i \Rightarrow e_i)_{i \in I} \mapsto [v_k/x_k]e_k}\ \textsf{step/cases/tag}$$

$$\frac{e_1 \mapsto e_1'}{k \cdot e_1 \mapsto k \cdot e_1'}\ \textsf{step/tag} \qquad \frac{e_0 \mapsto e_0'}{\textsf{case}\ e_0\ (i \cdot x_i \Rightarrow e_i)_{i \in I} \mapsto \textsf{case}\ e_0'\ (i \cdot x_i \Rightarrow e_i)_{i \in I}}\ \textsf{step/cases}_0$$

Recursive Types $\mu\alpha.\,\tau$

$$\frac{\Gamma, \alpha\ \mathit{type} \vdash \tau\ \mathit{type}}{\Gamma \vdash \mu\alpha.\,\tau\ \mathit{type}}\ \textsf{tp/mu}$$

$$\frac{\Gamma \vdash e : [\mu\alpha.\,\tau/\alpha]\tau}{\Gamma \vdash \textsf{fold}\ e : \mu\alpha.\,\tau}\ \textsf{tp/fold} \qquad \frac{\Gamma \vdash e : \mu\alpha.\,\tau}{\Gamma \vdash \textsf{unfold}\ e : [\mu\alpha.\,\tau/\alpha]\tau}\ \textsf{tp/unfold}$$

$$\frac{e\ \mathit{value}}{\textsf{fold}\ e\ \mathit{value}}\ \textsf{val/fold} \qquad \frac{v\ \mathit{value}}{\textsf{unfold}\ (\textsf{fold}\ v) \mapsto v}\ \textsf{step/unfold/fold}$$

$$\frac{e \mapsto e'}{\textsf{fold}\ e \mapsto \textsf{fold}\ e'}\ \textsf{step/fold} \qquad \frac{e \mapsto e'}{\textsf{unfold}\ e \mapsto \textsf{unfold}\ e'}\ \textsf{step/unfold}_0$$

Recursion

$$\frac{\Gamma, f : \tau \vdash e : \tau}{\Gamma \vdash \textsf{fix}\ f{:}\tau.\,e : \tau}\ \textsf{tp/fix} \qquad \frac{}{\textsf{fix}\ f.\,e \mapsto [\textsf{fix}\ f.\,e/f]e}\ \textsf{step/fix}$$