
CMSC 22100/32100: Programming Languages
Final Exam

M. Blume

December 11, 2008

1. (Well-founded sets and induction principles)

(a) State the *mathematical* induction principle and justify it informally.

15pt

If $P(0)$ and $\forall n \in \mathbb{N}. P(n) \Rightarrow P(n+1)$ then $\forall n \in \mathbb{N}. P(n)$.
Proof: Indirect: If the above is false, then there exists at least one k such that $\neg P(k)$. Pick the smallest such k and call it k_0 . Case $k_0 = 0$: Contradiction with $P(0)$. Case $k_0 = k_1 + 1$: Since k_0 is minimal, it must be that $P(k_1)$, but then we also have $P(k_1 + 1)$, i.e., $P(k_0)$, which is another contradiction.

(b) Let the pair (S, R) consists of some set S and a binary relation R over S (i.e., $R \subseteq S \times S$).

Given a subset $X \subseteq S$, what is a *minimal element* (with respect to R) of X ?

5pt

Element x is R -minimal in X if $\neg \exists y \in X. R(y, x)$.

(c) When is such a pair (S, R) considered *well-founded*?

5pt

$\forall X \subseteq S. X \neq \emptyset \Rightarrow \exists x \in X. x$ is minimal in X

- 5pt (d) Consider \mathbb{Z} , the set of integers. Give a relation $R_1 \subseteq \mathbb{Z} \times \mathbb{Z}$ such that (\mathbb{Z}, R_1) is not well-founded.

$R_1(x, y)$ iff $x < y$ under the usual less-than ordering $<$ on integers.

- 5pt (e) Give another relation R_2 such that (\mathbb{Z}, R_2) is well founded.

$R_1(x, y)$ iff $|x| < |y|$ (using $<$ on natural numbers).

- 10pt (f) Consider $(\mathbb{N} \times \mathbb{N}, R)$ where R is the lexicographic ordering, i.e., $R((x_1, y_1), (x_2, y_2))$ if and only if either $x_1 < x_2$ or else $x_1 = x_2$ and also $y_1 < y_2$. Show that this set-relation pair is well-founded.

Given a set X of pairs (x, y) , first pick a minimal element x_0 among the x -components. Then consider the restriction of X to x_0 , i.e., $\{(x_0, y) \mid (x_0, y) \in X\}$ and pick an element (x_0, y_0) such that y_0 is minimal among the y -components within the restriction. (x_0, y_0) is minimal in X as otherwise there would either have to be some (x_1, y_1) with $x_1 < x_0$, contradicting the choice of x_0 or an (x_0, y_2) with $y_2 < y_0$, contradicting the choice of y_0 .

- 10pt (g) Given a well-founded set-relation pair (S, R) , state the *principle of well-founded induction*.

$\forall P. (\forall x \in S. (\forall y \in S. R(y, x) \Rightarrow P(y)) \Rightarrow P(x)) \Rightarrow \forall x \in S. P(x)$

- (h) (*extra credit*) Justify the principle of well-founded induction informally.

20pt*

If the principle were not true, then the set of counterexamples $E = \{x \mid \neg P(x)\}$ would be non-empty. Choose a minimal $e \in E$ (which exists by well-foundedness). But being minimal in E means that $\forall y \in S. R(y, e) \Rightarrow P(y)$, which would imply $P(e)$ and, therefore, is a contradiction. Thus, E must be empty.

2. (*Terms, rules, and derivations*) Consider the following rules:

$$\frac{}{\mathbf{zero} \ \mathbf{nat}} \quad \mathbf{z} \qquad \frac{n \ \mathbf{nat}}{\mathbf{succ}(n) \ \mathbf{nat}} \quad \mathbf{s}$$

$$\frac{m \ \mathbf{nat}}{\mathbf{add}(\mathbf{zero}, m, m)} \quad \mathbf{ADD-Z} \qquad \frac{\mathbf{add}(n, m, s)}{\mathbf{add}(\mathbf{succ}(n), m, \mathbf{succ}(s))} \quad \mathbf{ADD-S-L}$$

- (a) State mathematically that the relation defined by the $\mathbf{add}(x, y, z)$ judgment is single-valued in its third argument.

5pt

If $\mathbf{add}(x, y, z)$ and also $\mathbf{add}(x, y, z')$ are derivable, then $z = z'$.

- (b) Prove the above statement about \mathbf{add} being single-valued.

15pt

By induction on x . Case $x = \mathbf{zero}$: Rule ADD-Z must have been used to derive both judgments, so $z = y = z'$. Case $x = \mathbf{succ}(\hat{x})$: Rules ADD-S-L must have been used to derive both judgments, so $z = \mathbf{succ}(\hat{z})$ and $z' = \mathbf{succ}(\hat{z}')$. Moreover, by inversion of ADD-Z we know $\mathbf{add}(\hat{x}, y, \hat{z})$ and also $\mathbf{add}(\hat{x}, y, \hat{z}')$. Thus, by IH: $\hat{z} = \hat{z}'$ from which we get $z = \mathbf{succ}(\hat{z}) = \mathbf{succ}(\hat{z}') = z'$.

- (c) Prove the following Lemma:

Lemma 1

If $\mathbf{add}(n, m, s)$ is derivable, then so is $\mathbf{add}(n, \mathbf{succ}(m), \mathbf{succ}(s))$.

15pt

By induction on n . Case $n = \mathbf{zero}$: By inversion of ADD-Z we have $m = s$, so $\mathbf{succ}(m) = \mathbf{succ}(s)$. Thus, by ADD-Z we also have $\mathbf{add}(\mathbf{zero}, \mathbf{succ}(m), \mathbf{succ}(s))$. Case $n = \mathbf{succ}(n')$: By inversion of ADD-S-L we have $s = \mathbf{succ}(s')$ with $\mathbf{add}(n', m, s')$. By IH, $\mathbf{add}(n', \mathbf{succ}(m), \mathbf{succ}(s'))$, so using ADD-S-L we find $\mathbf{add}(n, \mathbf{succ}(m), \mathbf{succ}(s))$.

- (d) Show a rule or an axiom that—if it were added to the set of rules and axioms shown above—invalidates Lemma 1. Explain how the added rule breaks your proof.

10pt

Sample rule:

$$\frac{}{\mathbf{add}(\mathbf{foo}, \mathbf{bar}, \mathbf{baz})}$$

The addition of the rule adds another case to the case analysis that is the heart of the proof. And, as it turns out, for case $n = \mathbf{foo}$ there is no way of concluding that $\mathbf{add}(\mathbf{foo}, \mathbf{succ}(\mathbf{bar}), \mathbf{succ}(\mathbf{baz}))$.

(e) We could render Lemma 1 as a rule:

$$\frac{\mathbf{add}(n, m, s)}{\mathbf{add}(n, \mathbf{succ}(m), \mathbf{succ}(s))} \text{ADD-S-R}$$

5pt

Is this rule *admissible* or *derivable*?

The rule is admissible, not derivable. (A derivable rule could not be broken by adding some other rule or axiom.)

3. (*Static and dynamic semantics, safety*) Consider the following variant of the Simply Typed λ -calculus (STLC):

types:	$\tau ::= \mathbf{nat} \mid \tau \rightarrow \tau$	
values:	$v ::= \mathbf{zero} \mid \mathbf{succ}(v) \mid \lambda x : \tau. e$	
expressions:	$e ::= x \mid$	variables
	$\mathbf{zero} \mid \mathbf{succ}(e) \mid$	zero and successor
	$\lambda x : \tau. e \mid e e \mid$	function abstraction and application
	$\mathbf{iter}(e, e)$	iteration

The $\mathbf{iter}(n, f)$ construct¹ computes the n -fold iteration of a function f . That is, to evaluate $\mathbf{iter}(e_1, e_2)$ we first evaluate e_1 to a number n and then e_2 to a function f of type $\tau \rightarrow \tau$ for some τ . The result is another function of type $\tau \rightarrow \tau$ which behaves like

$$\lambda x : \tau. \underbrace{f(f \dots (f x) \dots)}_{n \text{ times}}$$

(a) Give typing rules for this language. (The rules are the standard STLC rules plus one new rule for handling \mathbf{iter} .)

20pt

¹whose addition turns this language into a variant of *Gödel's T*, which is significantly more expressive than plain STLC

$$\begin{array}{c}
 \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash \mathbf{zero} : \mathbf{nat}} \\
 \\
 \frac{\Gamma \vdash v : \mathbf{nat}}{\Gamma \vdash \mathbf{succ}(v) : \mathbf{nat}} \qquad \frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1. e : \tau_1 \rightarrow \tau_2} \\
 \\
 \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau} \qquad \frac{\Gamma \vdash e_1 : \mathbf{nat} \quad \Gamma \vdash e_2 : \tau \rightarrow \tau}{\Gamma \vdash \mathbf{iter}(e_1, e_2) : \tau \rightarrow \tau}
 \end{array}$$

- (b) Give a *small-step* structural operational semantics, *i.e.*, show rules for deriving judgments of the relation $e \mapsto^1 e'$. Again, most of the rules are standard, but you have to handle **iter**. The trickiest rule, namely the one for **iter**(**succ**(n), f), is shown here:

20pt

$$\frac{(f = \lambda x : \tau. e)}{\mathbf{iter}(\mathbf{succ}(n), f) \mapsto^1 \lambda x : \tau. (\mathbf{iter}(n, f) (f x))}$$

Your task is to show all the *other* rules for the above language. Make sure the language is *call-by-value* and evaluates *from-left-to-right*.

$$\begin{array}{c}
 \frac{e_1 \mapsto^1 e'_1}{e_1 e_2 \mapsto^1 e'_1 e_2} \qquad \frac{e_2 \mapsto^1 e'_2}{v_1 e_2 \mapsto^1 v_1 e'_2} \\
 \\
 \frac{e_1 \mapsto^1 e'_1}{\mathbf{iter}(e_1, e_2) \mapsto^1 \mathbf{iter}(e'_1, e_2)} \qquad \frac{e_2 \mapsto^1 e'_2}{\mathbf{iter}(v_1, e_2) \mapsto^1 \mathbf{iter}(v_1, e'_2)} \\
 \\
 \frac{}{(\lambda x : \tau. e) v \mapsto^1 \{v/x\}e} \qquad \frac{}{\mathbf{iter}(\mathbf{zero}, \lambda x : \tau. e) \mapsto^1 \lambda x : \tau. x}
 \end{array}$$

- (c) Describe informally (in one sentence or two) what type safety means.

5pt

Small-step evaluation will not get stuck, *i.e.*, it either reaches a value or keeps going forever.

- (d) Which two lemmas, taken together, establish *type safety*? (What are the commonly used names of these lemmas?)

5pt

Progress and Preservation

- (e) State both of these two lemmas for the above language.

20pt

Progress: If $\emptyset \vdash e : \tau$ and e is not a value, then there exists some e' such that $e \mapsto^1 e'$.

Preservation: If $\emptyset \vdash e : \tau$ and $e \mapsto^1 e'$ then $\emptyset \vdash e' : \tau$.

15pt

(f) State the *canonical forms lemma* for this language.

- If $\emptyset \vdash v : \mathbf{nat}$ then $v = \mathbf{zero}$ or $v = \mathbf{succ}(v')$ for some v' .
- If $\emptyset \vdash v : \tau_1 \rightarrow \tau_2$ then $v = \lambda x : \tau_1. e$ for some x and some e .

15pt*

(g) (*Extra credit*) Show all changes to your small-step semantics from question 3b that are necessary to make it evaluate *from-right-to-left*.

$$\frac{e_2 \mapsto^1 e'_2}{e_1 e_2 \mapsto^1 e_1 e'_2} \qquad \frac{e_1 \mapsto^1 e'_1}{e_1 v_2 \mapsto^1 e'_1 v_2}$$

$$\frac{e_2 \mapsto^1 e'_2}{\mathbf{iter}(e_1, e_2) \mapsto^1 \mathbf{iter}(e_1, e'_2)} \qquad \frac{e_1 \mapsto^1 e'_1}{\mathbf{iter}(e_1, v_2) \mapsto^1 \mathbf{iter}(e'_1, v_2)}$$

4. (*Big-step semantics and substitution*) In question 3b we had the luxury (afforded by the small-step nature of the semantics) of being able to unwind $\mathbf{iter}(n, f)$ one step at a time. To construct an equivalent big-step semantics we need to come up with a way of generating $\lambda x : \tau. f (f \dots (f x) \dots)$ from n and f directly. For this we take advantage of the fact that f must have the form $\lambda x : \tau. e$.

Roughly, the idea is to substitute e “into itself” n times, so we define an *iterated version* of substitution. Let $\{e'/x\}e$ be ordinary substitution of e' for x in e . We write $\{e'/x\}^n e$ where n is a natural number of the form $\mathbf{succ}(\dots(\mathbf{succ}(\mathbf{zero}))\dots)$ to mean the n -times iterated version of substitution. The idea is to hold e' constant and to “grow” e by repeatedly substituting e' for x into it. The number of such iterations is given by n .

10pt

- (a) Give an inductive definition for $\{e'/x\}^n e$ in equational form.

$$\begin{aligned} \{e'/x\}^{\mathbf{zero}} e &= e \\ \{e'/x\}^{\mathbf{succ}(n)} e &= \{e'/x\}^n (\{e'/x\}e) \end{aligned}$$

- (b) Show the big-step rule for $\mathbf{iter}(e_1, e_2) \Downarrow v$ using iterated substitution. (Hint: Make sure you don't have any off-by-one errors, *i.e.*, be careful that $\mathbf{iter}(\mathbf{zero}, f)$ is always the identity function.)

10pt

$$\frac{e_1 \Downarrow n \quad e_2 \Downarrow \lambda x : \tau. e}{\mathbf{iter}(e_1, e_2) \Downarrow \lambda x : \tau. \{e/x\}^n x}$$

15pt

- (c) Give the remaining rules of the big-step operational semantics for the entire language.

$$\begin{array}{c}
 \frac{}{\mathbf{zero} \Downarrow \mathbf{zero}} \quad \frac{e \Downarrow n}{\mathbf{succ}(e) \Downarrow \mathbf{succ}(n)} \quad \frac{}{\lambda x : \tau. e \Downarrow \lambda x : \tau. e} \\
 \\
 \frac{e_1 \Downarrow \lambda x : \tau. e \quad e_2 \Downarrow v_2 \quad \{v_2/x\}e \Downarrow v}{e_1 e_2 \Downarrow v}
 \end{array}$$

5. (*Machines*) Big- and small-step semantics leave certain aspects of the mechanical execution of programs implicit, thereby being somewhat unrealistic when it comes to modelling how programs are actually implemented on real computers. Abstract machines make some of these aspects more explicit.

5pt

- (a) Which part of the semantics is being made explicit by the C machine, and what data structure is used to manifestly represent that part?

The C machine makes *control* explicit by representing the remainder of the computation as a stack of frames. The stack remembers any part of the computation that had to be postponed until after the current expression has been fully evaluated.

10pt

- (b) Which artificial aspect of the semantics is being *eliminated* by going from the C machine to the E machine? Which additional data structure is used there, and what does it represent?

The E machine eliminates substitutions by representing pending substitutions in a data structure called an *environment*. Environments are finite mappings from variables to (machine) values.

10pt

- (c) Which of the following language features can cause control frames to outlive the time when they are popped off the stack?

- exceptions (**try** ... **ow** ... and **fail**)
- first-class continuations (**letcc** and **throw**)
- higher-order functions
- mutable state (**ref**, **!** and **:=**)

continuations (and nothing else)

15pt*

6. (*Extra credit*) We have seen that many type constructors “missing” from System F can be simulated via Church-encodings. Examples for this included natural numbers, booleans, general product and sum types, inductive types such as lists or trees, and even existentials.

Notably absent from this list are recursive types. Do you think that this is just a coincidental omission, or is there a fundamental reason why general recursive types cannot be Church-encoded in pure System F?

If you think that they can be simulated, try to sketch out an encoding. (Your encoding does not need to be complete. For example, you don't have to show the encoding of **roll** and **unroll**. Showing the encoding of $\mu\alpha.\tau$ would suffice.) If, on the other hand, you think that this is not possible, give an informal proof of this fact (using your knowledge of other

facts regarding System F).

It is not possible to Church-encode recursive types. All programs in pure System F terminate, but the addition of recursive types makes the language Turing-complete. Church-encoded types are still plain System F types, and they cannot raise the expressive power of the language, so recursive types, whose addition does raise the expressive power, are out of reach.