CS131 Typed Lambda Calculus Worksheet

Due Thursday, April 19th

Name: ______________________________

CAS ID (e.g., abc01234@pomona.edu): ______________________________

I encourage you to collaborate. Please record your collaborations below.

Each question is worth one point, except problems marked with a (C) are "challenge" problems—go ahead and test your mettle, but these are longer or harder than anything I'd put on an exam and worth no course credit.

Please turn in your work as a printout of this sheet, not on separate paper. If you would rather typeset your work, I can give you the LaTeX... but you'll learn more by writing it by hand.

Collaborators: ______________________________

1 Lambda calculus with booleans

$$\begin{array}{rcl} t & ::= & \mathsf{bool} \mid t_1 \to t_2 \\ e & ::= & x \mid e_1\ e_2 \mid \lambda x{:}t.\ e \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\ \Gamma & ::= & \cdot \mid \Gamma, x{:}t \end{array}$$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t} \qquad \frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1\ e_2 : t_2} \qquad \frac{\Gamma, x{:}t_1 \vdash e : t_2}{\Gamma \vdash \lambda x{:}t_1.\ e : t_1 \to t_2}$$

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{bool} \qquad \Gamma \vdash e_2 : t \qquad \Gamma \vdash e_3 : t}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : t}$$

1.1 Type hunting

For each term e, find a context Γ and type t that makes that term well typed, i.e., $\Gamma \vdash e : t$.

1. $\Gamma \vdash \mathsf{if}\ x\ \mathsf{then}\ x\ \mathsf{else}\ y : t$ $\Gamma =$ ____________ $t =$ ____________

2. $\Gamma \vdash x\ y : t$ $\Gamma =$ ____________ $t =$ ____________

3. $\Gamma \vdash \lambda x{:}\mathsf{bool}.\ x : t$ $\Gamma =$ ____________ $t =$ ____________

4. $\Gamma \vdash \lambda x{:}\mathsf{bool}\to\mathsf{bool}.\ y\ x\ \mathsf{true} : t$ $\Gamma =$ ____________ $t =$ ____________

5. $\Gamma \vdash \lambda x{:}\mathsf{bool}\to\mathsf{bool}.\ y\ (x\ \mathsf{true}) : t$ $\Gamma =$ ____________ $t =$ ____________

6. $\Gamma \vdash \lambda x{:}t_1.\ \mathsf{if}\ y\ \mathsf{then}\ x\ \mathsf{else}\ y : t$ $\Gamma =$ ____________ $t =$ ____________

   $t_1 =$ ____________

7. $\Gamma \vdash \lambda x{:}t_1\ y{:}t_2\ z{:}\mathsf{bool}.\ \mathsf{if}\ z\ \mathsf{then}\ x\ \mathsf{else}\ y : t$ $\Gamma =$ ____________ $t =$ ____________

   $t_1 =$ ____________ $t_2 =$ ____________

1.2 Term hunting

For each type t, find a closed term e that has that type, i.e., $\cdot \vdash e : t$.

1. $\cdot \vdash e : \mathsf{bool} \rightarrow \mathsf{bool}$

2. $\cdot \vdash e : \mathsf{bool} \rightarrow \mathsf{bool} \rightarrow \mathsf{bool}$

3. $\cdot \vdash e : (\mathsf{bool} \rightarrow \mathsf{bool}) \rightarrow \mathsf{bool}$

4. $\cdot \vdash e : (\mathsf{bool} \rightarrow \mathsf{bool} \rightarrow \mathsf{bool}) \rightarrow \mathsf{bool}$

1.3 A failed hunt

Explain why there are no t_1, t_2, and t_3 such that $\cdot \vdash (\lambda x{:}t_1.\ x\ x)\ (\lambda x{:}t_2.\ x\ x) : t_3$.

1.4 A type you can count on

How many semantically different closed values are there of type bool? That is, we can write an infinite number of closed programs with type bool, but how many different values can we get out? List them as lambda calculus terms.

How many semantically different values are there of type bool→bool? There are infinitely many *syntactically* different values of type bool→bool, but many of them behave the same. How many different behaviors can a value typed at bool→bool exhibit? List them as (typed) lambda calculus terms.

1.5 We make our own rules, here

Suppose we extended our grammar with a forms $e_1 \land e_2$ (conjunction, read "e_1 and e_2"), $e_1 \lor e_2$ (disjunction, read "e_1 or e_2"), and $\neg e$ (negation, read "not e").

Write typing rules for these forms.

1.6 Conditional love (C)

Suppose we extend our grammar with a multi-branch conditional, of the form:

$$\mathsf{cond}\ \{e_{11} \Rightarrow e_{12}; e_{21} \Rightarrow e_{22}; \ldots; \_ \Rightarrow e_d\}$$

Here are small-step evaluation rules for it:

$$\frac{e_{11} \longrightarrow e'_{11}}{\mathsf{cond}\ \{e_{11} \Rightarrow e_{12}; e_{21} \Rightarrow e_{22}; \ldots; \_ \Rightarrow e_d\} \longrightarrow \mathsf{cond}\ \{e'_{11} \Rightarrow e_{12}; e_{21} \Rightarrow e_{22}; \ldots; \_ \Rightarrow e_d\}}$$

$$\frac{}{\mathsf{cond}\ \{\mathsf{true} \Rightarrow e_{12}; e_{21} \Rightarrow e_{22}; \ldots; \_ \Rightarrow e_d\} \longrightarrow e_{12}}$$

$$\frac{}{\mathsf{cond}\ \{\mathsf{false} \Rightarrow e_{12}; e_{21} \Rightarrow e_{22}; \ldots; \_ \Rightarrow e_d\} \longrightarrow \mathsf{cond}\ \{e_{21} \Rightarrow e_{22}; \ldots; \_ \Rightarrow e_d\}}$$

$$\frac{}{\mathsf{cond}\ \{\_ \Rightarrow e_d\} \longrightarrow e_d}$$

In English, a multi-branch conditional evaluates each of its branches $e_{i1} \Rightarrow e_{i2}$ in turn; if e_{i1} yields true, then it executes e_{i2}; otherwise, it keeps checking other branches. If none of the branches match, it runs the default branch e_d.

Write a typing rule for cond.

2 Tuples

2.1 Two's company

Write the typing rules for a lambda calculus extended with pair types (t_1, t_2), pairs (e_1, e_2) and projections $\mathsf{fst}\ e$ and $\mathsf{snd}\ e$.

2.2 The trouble with triples

Write the typing rules for a lambda calculus extended with *triples*, i.e., the type (t_1, t_2, t_3) and the terms (e_1, e_2, e_3), $\mathsf{first}\ e$, $\mathsf{second}\ e$, and $\mathsf{third}\ e$.

2.3 It's a twofer (C)

Devise a syntactic sugar for encoding triples in terms of pairs. That is, write down four pieces of syntactic sugar that take the triple type and expression syntax of Problem 2.2 to a program in the pair syntax of Problem 2.1. Make sure you syntactic sugar: (a) has the right behavior; and (b) preserves types appropriately, i.e., if (e_1, e_2, e_3) is well typed per your rules in Problem 2.2, its encoding should be well typed per your rules in Problem 2.1.

2.4 No limits (C)

Write typing rules for tuples of arbitrary length, i.e., types $(t_1, \ldots, t_n)$, tuples $(e_1, \ldots, e_n)$, and projections $\pi_i\ e$ which get the ith element of a tuple. Be sure to allow n to be 0.

3 Other extensions

3.1 List of demands

Suppose we have integers (type int) in the lambda calculus. Add lists of integers to the simply typed lambda calculus, i.e., a type $\mathsf{intlist}$ and terms nil, $\mathsf{cons}\ e_1\ e_2$, and $\mathsf{case}\ e_1\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow e_2; \mathsf{cons}\ x_1\ x_2 \Rightarrow e_3\}$. Here are evaluation rules for case:

$$\frac{e_1 \longrightarrow e_1'}{\mathsf{case}\ e_1\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow e_2; \mathsf{cons}\ x\ y \Rightarrow e_3\} \longrightarrow \mathsf{case}\ e_1'\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow e_2; \mathsf{cons}\ x\ y \Rightarrow e_3\}}$$

$$\frac{}{\mathsf{case}\ \mathsf{nil}\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow e_2; \mathsf{cons}\ x\ y \Rightarrow e_3\} \longrightarrow e_2}$$

$$\frac{}{\mathsf{case}\ (\mathsf{cons}\ v_1\ v_2)\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow e_2; \mathsf{cons}\ x_1\ x_2 \Rightarrow e_3\} \longrightarrow e_3[{}^{v_1}/_{x_1}][{}^{v_2}/_{x_2}]}$$

Write typing rules for nil, $\mathsf{cons}\ e_1\ e_2$, and $\mathsf{case}\ e_1\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow e_2; \mathsf{cons}\ x_1\ x_2 \Rightarrow e_3\}$.

This problem's title was inspired by the inimitable Saul Williams (see `https://www.youtube.com/watch?v=zDMtaIcrfQ0`).

3.2 Sum more than others

Add the Haskell `Either` datatype to the simply typed lambda calculus. That is, extend the rules of the simply typed lambda calculus to include so-called *sum* types $t_1 + t_2$ and terms $\mathsf{left}\ e$, $\mathsf{right}\ e$, and a pattern matching form like $\mathsf{case}\ e_1\ \mathsf{of}\ \{\mathsf{left}\ x_1 \Rightarrow e_2; \mathsf{right}\ x_2 \Rightarrow e_3\}$. You'll need to write typing rules for each of these new syntactic forms.

What changes would you make, if any, to implement these rules in a type checker?

Fun fact: pairs are also called *product* types, and are sometimes written $t_1 \times t_2$ or $t_1 * t_2$ to emphasize this fact. The sum/product analogies are why datatypes are sometimes called *algebraic* datatypes.

3.3 Get your fix

In HW07, we introduced recursion by adding let rec. We could have instead added recursive functions directly. Suppose we extend the simply typed lambda calculus with an expression form $\mathsf{fix}\ f(x{:}t_1) : t_2 = e$ Here f is the name of the function, x is its argument and t_1 is the argument's type, t_2 is the return type, and e is the body. It evaluates as follows:

$$\frac{}{(\mathsf{fix}\ f(x{:}t_1) : t_2 = e)\ v \longrightarrow e[{}^{v}/_{x}][{}^{\mathsf{fix}\ f(x:t_1)\ :\ t_2\ =\ e}/_{f}]}$$

For example,

$$(\mathsf{fix}\ \mathsf{fact}(n{:}\mathsf{int}) : \mathsf{int} = \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n * \mathsf{fact}\ (n-1))\ 5 \to^{*} 120$$

Write a typing rule for fix.

3.4 That's an order (C)

Suppose we have a simply typed lambda calculus with $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$. Let $e_1; e_2$ be syntactic sugar for $\mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2$, i.e., it runs e_1, throws away the result, and then runs e_2.

Let's extend the simply typed lambda calculus with state, i.e., a type $\mathsf{ref}\ t$ and terms $\mathsf{new}\ e$ (which allocates a new reference with e as its initial value), $\mathsf{read}\ e_1$ (which looks up the current value of the reference in e_1), and $\mathsf{write}\ e_1\ e_2$ (which sets the reference in e_1 to have a new value in e_2. For example, we have:

$$\mathsf{let}\ x = \mathsf{new\ true\ in\ read}\ x \longrightarrow^* \mathsf{true}$$

$$\mathsf{let}\ x = \mathsf{new\ true\ in\ write}\ x\ \mathsf{false} \longrightarrow^* ()$$

$$\mathsf{let}\ x = \mathsf{new\ true\ in\ write}\ x(\neg(\mathsf{read}\ x));\ \mathsf{read}\ x \longrightarrow^* \mathsf{false}$$

Write typing rules for new, read, and write. Don't allow "strong updates", which change the type of a variable. That is, $\mathsf{let}\ x = \mathsf{new\ true\ in\ write}\ x\ (\lambda x{:}\mathsf{bool}.\ x)$ should be ill typed. You can assume that there is a rule saying $\Gamma \vdash () : ()$.

Write small-step reduction rules for these features. Note that you'll need to add something to the step relation to keep track of the values of each reference. (How did we handle mutation for the While language?) You might need another page.