What to turn in

Turn in the written parts of the assignment during class on the due date. For the programming part, you should mail your version of the file cps-translate.sml to `bwh6@cs.cornell.edu` by 5PM on that day.

Problem 1: Lambda Calculus Dualities [30 pts]

Where duality is transcended peace reigns. — Zen proverb

This problem explores dualities and correspondences between λ-calculus expressions and continuations, and between CBV and CBN semantics. Consider the following lambda calculus with sums and products:

$$e \;::=\; x \mid \langle e_1, e_2\rangle \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{case}\ e_1\ \mathsf{of}\ (e_2 \,|\, e_3) \mid \lambda x.\, e \mid e_1\ e_2$$

As usual, $\langle e_1, e_2\rangle$ constructs a pair, and fst and snd destruct it; similarly, inl and inr construct a sum and $\mathsf{case}\ e_1\ e_2\ e_3$ destructs it. This language doesn't make dualities explicit. Therefore, we choose to refactor our source calculus, in order to make dualities explicit in the syntax. We syntactically separate expressions from the source calculus into *terms*, which produce a value then cease execution, and *coterms*, which consume a value then continue execution. We're already familiar with coterms by another name: *continuations*. Our target calculus is described by the following grammar.

$$\begin{array}{lcl} S & ::= & P \rhd C \\ P & ::= & x \mid \langle P_1, P_2\rangle \mid \mathsf{inl}\ P \mid \mathsf{inr}\ P \mid \mathsf{not}\ C \mid \lambda x.\, P \mid \alpha.(S) \\ C & ::= & \alpha \mid [C_1, C_2] \mid \mathsf{fst}\ C \mid \mathsf{snd}\ C \mid \mathsf{not}\ P \mid P \mathbin{@} C \mid x.(S) \end{array}$$

The three syntactic classes in the grammar are statements S, producers (or terms) P, and consumers (coterms or continuations) C. We'll also use T, Q, and D, respectively, for these three classes. Statements are the plumbing by which values are passed from terms to coterms: a term produces a value, and passes it to the continuation represented by the coterm.

Since we have both terms and coterms, we must have both variables and covariables. Variables are written with Roman letters (e.g., x) and hold values. Covariables are written with Greek letters (e.g., α) and represent continuations.

In addition to variables, terms consist of pairs $\langle P_1, P_2\rangle$, injections $\mathsf{inl}\ P$ and $\mathsf{inr}\ P$, complements of coterms, function abstractions, and covariable abstractions. Most of these are self-explanatory — evaluation of a term consists of evaluating its subterms then producing a value, such as a pair, an injection, or a function. The complement $\mathsf{not}\ C$ changes a coterm C into a term by treating the continuation C as a value. Finally, a covariable abstraction $\alpha.(S)$ binds a free covariable α to the current continuation in the body of the statement S. Hence, the evaluation of $\alpha.(S)$ consists of evaluating S, then producing the value that is passed to α during the execution of S.

Coterms consist of covariables, sums, projections $\mathsf{fst}\ C$ and $\mathsf{snd}\ C$, complements of terms, function applications, and variable abstractions. Evaluation of covariables has already been described. Evaluating a sum $[C_1, C_2]$ is like evaluating a case expression: it consumes a value, then if the value is a left injection, passes the injected value to continuation C_1, or if it is a right injection, C_2. The projection $\mathsf{fst}\ C$ consumes a pair, projects out the first component, then passes it to continuation C; $\mathsf{snd}\ C$ is similar. The coterm $\mathsf{not}\ P$ changes a term P into a coterm by consuming a value representing a continuation and passing to it the value produced by P. A function application $P \mathbin{@} C$ consumes an value representing function , evaluates P, applies the function to the value produced by evaluating P, then passes the value produced by the application to the continuation C. Finally, a variable abstraction $x.(S)$ is evaluated by consuming a value, binding it to x, then executing S.

Here is an example program. Suppose we want a program that, when applied to $\langle \mathsf{inl}\ a, b\rangle$ returns $\langle a, b\rangle$, and when applied to $\langle \mathsf{inr}\ a, b\rangle$ returns $\langle b, a\rangle$. Suppose that the free variable x contains the input pair. Then in the source calculus, one such program would be:

$$\mathsf{case}\ (\mathsf{fst}\ x)\ \mathsf{of}\ (\lambda y.\, \langle y, \mathsf{snd}\ x\rangle \mid \lambda y.\, \langle \mathsf{snd}\ x, y\rangle)$$

In the target calculus, a similar program could be written:

$$x \rhd \mathsf{fst}\ [y.(\langle y, \beta.(x \rhd \mathsf{snd}\ \beta)\rangle \rhd \alpha), y.(\langle \beta.(x \rhd \mathsf{snd}\ \beta), y\rangle \rhd \alpha)]$$

Here, α is the continuation enclosing the whole expression. The above expression takes a value x, passes it to a continuation which applies fst, and then passed the result to one of the two continuations in the sum $[C_l, C_r]$. The left continuation binds y to the value it receives (in the example with $x = \langle \mathsf{inl}\ a, b\rangle$, variable y would be a), builds up a pair and passes it to the current continuation α. The pair has y on the first position, and the second component of x in the second position, which is expressed as $\beta.(x \rhd \mathsf{snd}\ \beta)$. In English, this is a value obtained by labeling the current continuation β and passing x to a continuation which applies snd, then passes the result to β.

a) We want to give a translational semantics from the source calculus to the target calculus using a translation function $\mathcal{T}[\![e]\!]\alpha$, where e is a piece of source syntax and α is the current continuation. Some of the rules are as follows:

$$\mathcal{T}[\![x]\!]\alpha = x \rhd \alpha \qquad \mathcal{T}[\![\langle e_1, e_2\rangle]\!]\alpha = \langle \beta.(\mathcal{T}[\![e_1]\!]\beta), \beta.(\mathcal{T}[\![e_2]\!]\beta)\rangle \rhd \alpha$$

$$\mathcal{T}[\![\mathsf{fst}\ e]\!]\alpha = \beta.(\mathcal{T}[\![e]\!]\beta) \rhd \mathsf{fst}\ \alpha \qquad \mathcal{T}[\![\mathsf{snd}\ e]\!]\alpha = \beta.(\mathcal{T}[\![e]\!]\beta) \rhd \mathsf{snd}\ \alpha$$

Note that the translation is written for arbitrary expressions e, e_1, e_2. In particular cases such as $\mathsf{fst}\ x$, the translation yields: $\beta.(x \rhd \beta) \rhd \mathsf{fst}\ \alpha$. This can be further reduced (in one step) to $x \rhd \mathsf{fst}\ \alpha$. The example above uses this simplified form.

Write the remaining translation rules.

b) We next want to define the CBV and CBN small-step operational semantics for the target calculus, according to the informal description at the beginning of this problem. These should be two relations $\to_V$ and $\to_N$ over $S \times S$. The evaluations reduce statements $P \rhd C$ to simpler statements $P' \rhd C'$. Your rules can assume that reductions may be applied in any order, similar to the full β-reduction in λ-calculus. Since there are both terms and coterms in the target calculus, we define both values V and covalues Z, as follows:

$$\begin{array}{rcl} V & ::= & x \mid \langle V, V\rangle \mid \mathsf{inl}\ V \mid \mathsf{inr}\ V \mid \mathsf{not}\ C \mid \lambda x.\, P \\ Z & ::= & \alpha \mid [Z, Z] \mid \mathsf{fst}\ Z \mid \mathsf{snd}\ Z \mid \mathsf{not}\ P \mid P@Z \end{array}$$

Under CBV, the evaluation of a statement $P \rhd C$ first reduces P to a value V, then passes it to continuation C; or, if P reduces to a covariable abstraction $\alpha.(S)$, then $P \rhd C$ reduces to S with α bound to C. Under CBN, the evaluation of $P \rhd C$ first reduces C to a covalue Z, then passes the unevaluated term P to Z; or, if C reduces to a variable abstraction $x.(S)$, then $P \rhd C$ reduces to S with x bound to P.

We provide the CBV and CBN evaluation rules for several expressions:

$$\begin{array}{rcl@{\qquad\qquad}rcl} \langle V, W\rangle \rhd \mathsf{fst}\ C & \to_V & V \rhd C & \langle P, Q\rangle \rhd \mathsf{fst}\ Z & \to_N & P \rhd Z \\ \mathsf{not}\ C \rhd \mathsf{not}\ P & \to_V & P \rhd C & \mathsf{not}\ C \rhd \mathsf{not}\ P & \to_N & P \rhd C \\ \lambda x.\, P \rhd V@C & \to_V & V \rhd x.(P \rhd C) & \lambda x.\, P \rhd Q@Z & \to_N & Q \rhd x.(P \rhd Z) \end{array}$$

Write the remaining rules for CBN and CBV evaluation in the target calculus.

c) From the syntax of the target language, there is a straightforward duality between terms and coterms. We assume that each variable x is uniquely associated with a covariable x°, and similarly for α and α°. We define the duality operator for a fragment of the target calculus, as follows.

$$\begin{array}{rcl@{\qquad\qquad}rcl}
(x)^\circ & \equiv & x^\circ & (\alpha)^\circ & \equiv & \alpha^\circ \\
\langle P, Q\rangle^\circ & \equiv & [P^\circ, Q^\circ] & [C, D]^\circ & \equiv & \langle C^\circ, D^\circ\rangle \\
(\mathsf{inl}\ P)^\circ & \equiv & \mathsf{fst}\ P^\circ & (\mathsf{fst}\ C)^\circ & \equiv & \mathsf{inl}\ C^\circ \\
(\alpha.(S))^\circ & \equiv & \alpha^\circ.(S^\circ) & (x.(S))^\circ & \equiv & x^\circ.(S^\circ)
\end{array}$$

where the duality of statements is: $(P \rhd C)^\circ \equiv C^\circ \rhd P^\circ$. Prove that $\to_V$ and $\to_N$ are duals: $S \to_V T$ iff $S^\circ \to_N T^\circ$. You have to prove this just for the fragment of the target calculus shown above.

d) Assuming that the target language has been extended with standard types, what do the relations $\langle P, Q\rangle^\circ \equiv [P^\circ, Q^\circ]$ and $P^{\circ\circ} = P$ correspond to under the Curry-Howard isomorphism? Explain briefly.

Problem 2: Java Bytecodes [25 pts]

Consider the following syntax which describes a small subset of the possible bytecode instructions in Java:

$$\begin{array}{rcl}
s & ::= & \mathsf{bipush}\ n \ \mid\ \mathsf{iload}\ m \ \mid\ \mathsf{istore}\ m \ \mid\ \mathsf{iadd} \ \mid\ \mathsf{ifcond}\ i \\
 & & \mathsf{new} \ \mid\ \mathsf{aload}\ m \ \mid\ \mathsf{astore}\ m \ \mid\ \mathsf{if_acmpcond}\ i \\
n, m, i & \in & \mathbb{N}
\end{array}$$

A program P is a sequence of n_P such statements. We denote by s_i the statement representing the i^{th} element in this sequence. Intuitively, i is a program point, and s_i is the instruction at that point. Hence, i models the continuation at that point in the program.

The execution of a program takes place on an abstract stack machine. The bytecode instructions reads values from the stack and update the stack with new values. It also reads and updates variable values. A configuration $\langle i, \sigma, \rho\rangle$ in the operational semantics of this language consists of a program counter i, a stack of values σ, and an environment ρ which holds values for variables. A final configuration is one where the execution reaches the point after the last instruction: $i = n_P + 1$. The values in this language are integers and references to objects. The language models variables as numbers m, so an environment ρ is a map from integers to the values of the corresponding variables.

The rules below give the operational semantics for the instructions in this language. Informally, the instructions push constants on the stack (bipush), create new objects (new), load and store integers and references from variables to the stack and vice-versa (iload, istore, aload, astore), or add integers (iadd). The control-flow constructs include branch instructions which test integers for zero (ifcond) or compare two references (if_acmpcond). We use the notation $\sigma :: v$ for stack concatenation; the element v is the top element of the resulting stack. We denote by m variable indices, by i, j program points, by n integer constants, and by l object references.

$$\frac{s_i \equiv \mathsf{bipush}\ n}{\langle i, \sigma, \rho\rangle \to \langle i+1, \sigma :: n, \rho\rangle} \qquad \frac{s_i \equiv \mathsf{iload}\ m \quad \rho(m) = n}{\langle i, \sigma, \rho\rangle \to \langle i+1, \sigma :: n, \rho\rangle} \qquad \frac{s_i \equiv \mathsf{istore}\ m}{\langle i, \sigma :: n, \rho\rangle \to \langle i+1, \sigma, \rho[m \mapsto n]\rangle}$$

$$\frac{s_i \equiv \mathsf{iadd} \quad n = n_0 + n_1}{\langle i, \sigma :: n_0 :: n_1, \rho\rangle \to \langle i+1, \sigma :: n, \rho\rangle} \qquad \frac{s_i \equiv \mathsf{aload}\ m \quad \rho(m) = l}{\langle i, \sigma, \rho\rangle \to \langle i+1, \sigma :: l, \rho\rangle} \qquad \frac{s_i \equiv \mathsf{astore}\ m}{\langle i, \sigma :: l, \rho\rangle \to \langle i+1, \sigma, \rho[m \mapsto l]\rangle}$$

$$\frac{s_i \equiv \mathsf{new} \quad \textit{fresh}\ l}{\langle i, \sigma, \rho\rangle \to \langle i+1, \sigma :: l, \rho\rangle} \qquad \frac{s_i \equiv \mathsf{ifcond}\ j \quad n \neq 0}{\langle i, \sigma :: n, \rho\rangle \to \langle j, \sigma, \rho\rangle} \qquad \frac{s_i \equiv \mathsf{ifcond}\ j \quad n = 0}{\langle i, \sigma :: n, \rho\rangle \to \langle i+1, \sigma, \rho\rangle}$$

$$\frac{s_i \equiv \mathsf{if\_acmpcond}\ j \quad l_0 \neq l_1}{\langle i, \sigma :: l_0 :: l_1, \rho\rangle \to \langle j, \sigma, \rho\rangle} \qquad \frac{s_i \equiv \mathsf{if\_acmpcond}\ j \quad l_0 = l_1}{\langle i, \sigma :: l_0 :: l_1, \rho\rangle \to \langle i+1, \sigma, \rho\rangle}$$

We want to derive typing rules for these bytecode instructions. The problem that we encounter is that these operations expect their specific values in two places: on the stack and in the local variables. Hence, type checking instructions requires type checking both stacks and variables. We define *typed bytecodes* to be bytecodes plus:

1. a type environment $\Gamma : \mathbb{N} \to \{\mathsf{int}, \mathsf{ref}\}$ for variables. The type of variable n is $\Gamma(n)$.

2. a type environment $\Delta : \mathbb{N} \to \mathit{StackTypes}$ for stacks, which provides types for stacks each program point[1]. The type of the stack at program point i is $\Delta(i)$. A stack type $\tau_\sigma \in \mathit{StackTypes}$ is either an empty stack or an arbitrary stack, with a number of specific primitive types int or ref at the top:

$$\begin{array}{lcl} \tau_\sigma & ::= & \mathsf{empty} \;\mid\; \mathsf{any} \;\mid\; \tau_\sigma :: \tau_p \\ \tau_p & ::= & \mathsf{int} \;\mid\; \mathsf{ref} \end{array}$$

The typing rules for each instruction s_i are of the form $\Gamma, \tau_\sigma \vdash s, \tau'_\sigma$, with the meaning that starting in a state where the stack has type τ_σ and where variable have the types stored in Γ, the execution of statement s_i will produce a new stack with type τ'_σ. Once this relation is defined, one can type check a program P by checking if Δ correctly characterize the possible types at each point. We write $\Gamma, \Delta \vdash P$ if the program P type-checks in the typing environments Γ and Δ:

$$\frac{\forall i, j \;.\; \Gamma, \Delta(i) \vdash s_i, \tau'_\sigma \wedge flow(i,j) \Rightarrow \tau'_\sigma \leq \Delta(j)}{\Gamma, \Delta \vdash P}$$

where *flow(i,j)* indicates that there is flow of control form i to j: either $j = i + 1$ or instruction s_i is a conditional jump with target point j. Also, we use the notation $\tau_1 \leq \tau_2$ for stack types to indicate that the top of the stack type τ_1 matches the top of the stack type τ_2. For instance, $\mathsf{any} :: \mathsf{ref} :: \mathsf{int} \leq \mathsf{any} :: \mathsf{int}$.

a) Write the typing rules $\Gamma, \tau_\sigma \vdash s, \tau'_\sigma$ for each bytecode instruction s.

b) Define a satisfiability relation $\sigma \models \tau_\sigma$ between stacks and stack types, and a relation $\rho \models \Gamma$ between variable environments and type environments. Then use these relations to formally express the soundness condition for a program P.

c) Formally express and prove the Type Preservation property, restricting your proof to the following instructions: iadd, astore, and ifcond.

d) Formally express and prove the Progress property, restricting your proof to bipush, iload, and if_acmpcond.

Problem 3: Recursive Types [10 pts]

a) Once we add recursive types in the typed lambda calculus, the untyped lambda calculus becomes a subset of the typed lambda calculus, where each variable x has type $\mu X.X \to X$. Show this fact, using a translation $\mathcal{T}[\![e]\!]$ which maps an untyped lambda term to its equivalent typed version, using the $\mathsf{fold}[\mu X.\tau]\ e$ and $\mathsf{unfold}\ e$ operators. The untyped calculus is the standard one: $e ::= x \mid \lambda x.\, e \mid e_0 e_1$.

b) The above translation allows us to produce a typed version of the Y combinator, and thus allows us to express recursive function. However, since all the variables have type $\mu X.X \to X$, all functions have type $(\mu X.X \to X) \to (\mu X.X \to X)$, and the Y combinator only finds fixed points for such function types. Write a more general typed fixed point combinator Y_τ which computes fixed points for arbitrary functions of type $\tau \to \tau$.

[1] In real Java systems, the bytecode verifier computes such stack types at each point.

Problem 4: Closure Conversion and CPS [35 pts]

In this problem you will translate a simple call–by–value language into a target language that is very close to assembly language. Hence, you will be exploring some of the transformations needed for compilation to assembly.

The key feature of the target language is that lambda terms in the language cannot have any free variables other than the argument variable itself. This makes sense because in a low-level assembly language it is easy to refer to local variables, but it is difficult to access variables from containing lexical scopes. The process of removing free variables from lambda terms is known as *closure conversion* or *lambda lifting* (because once your lambda terms have no free variables, they can be lifted to the top level of the program).

The source language here is a strict, call–by–value λ-calculus language. To simplify writing examples, it also adds built–in boolean constants and an if expression. Its syntax is:

$$\begin{array}{rcl} op & ::= & + \mid - \mid * \\ e & ::= & x \mid n \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{iszero}\ e \mid \mathsf{if}\ e_0\ e_1\ e_2 \mid \mathsf{lam}\ x\ e \mid e_1\ e_2 \mid op\ e_1\ e_2 \end{array}$$

The operators $+$, $*$, and $-$ are strictly binary, and iszero is unary, producing either true or false. We have provided a parser for this language, the function LambdaSimpleParser.parse, which should make it easy to write test cases for your translation and turn them into LambdaSimpleAST.exp's.

The syntax of the target language is:

$$\begin{array}{rcl} op & ::= & + \mid - \mid * \\ e & ::= & x \mid \mathsf{lam}\ x\ s\ (\text{where } FV[\![s]\!] \subseteq \{x\}) \mid n \mid x_1\ op\ x_2 \mid \mathsf{halt} \mid \langle x_1, ..., x_n \rangle \mid x[n] \\ s & ::= & x_1\ x_2 \mid \mathsf{let}\ x = e\ \mathsf{in}\ s \mid \mathsf{ifz}\ x\ s_1\ s_2 \end{array}$$

Here, $\langle x_1, ..., x_n \rangle$ represents a "tuple" of n elements; if $y = \langle x_1, ..., x_n \rangle$ then $y[i] \to x_i$. The application of x to halt (e.g. let $h =$ halt in $h\ x$) terminates execution, returning the value bound to x. Applications are restricted to be of one variable to another variable, and abstractions $\lambda x.\, s$ are restricted to terms s with at most one free variable, necessarily x. The ifz statement tests whether the variable x is zero and executes s_1 if so, s_2 otherwise. Note that the target language does not have boolean values. You will represent booleans as integers, translating true as 0 and false as 1.

Note that the statements s in a let construct let $x = e$ in s in the target language are essentially continuations. They contain the "rest of the program" to execute after the assignment $x = e$ has been performed. We therefore refer to the target language as a CPS language. For example, the expression $(+\ (*\ 4\ 3)\ 5)$ would need to be translated into the target language as something like:

$$\begin{array}{l} \mathsf{let}\ x_1 = 4 * 3\ \mathsf{in} \\ \mathsf{let}\ x_2 = x_1 + 5\ \mathsf{in} \\ s \end{array}$$

where x_1 and x_2 are new temporary variables, and s is the rest of the computation that needs to happen after the expression $(+\ (*\ 4\ 3)\ 5)$ has been evaluated.

Also note that there is a strong correspondence between statements in the CPS language and the instructions of an ordinary machine language. Each statement in the output code roughly corresponds to one assembly instruction. The only exception is when a let construct binds a variable to an abstraction — that corresponds to a function declaration in the assembly code.

a) Define the translation function $\mathcal{T}[\![e]\!]$, where e is an expression of the source language, and the result is a statement of the target language.

   The main hurdle to overcome is that whereas the source language has no restrictions on non-local variables, the target language only allows function bodies to have one free variable. You will need to find some way of packing a function's real argument, its return continuation, and any other necessary information into the single argument to the CPS function. In addition, note that source–language function values capture the values of their free variables (lexically), whereas target–language function values cannot. Therefore, you will need to translate source–language function values to something other than a target–language function value!

We suggest that you separate your translation into two phases. The first phase will perform the CPS conversion; it will translate source expressions into the target language, allowing free variables in function bodies. The second pass will then perform the closure conversions, packing together multiple function arguments.

b) Implement your translation function(s) from part (a). You are allowed to modify only the file "cps-translate.sml". To see some of the parsing and translation functions being used, look in "top.sml" for some examples. If you chose to perform CPS and closure conversion in two separate passes, note that the evaluator does not know how to handle lambda terms that have free variables.

Your translation should be a function cps_translate : LambdaSimpleAst.exp → CPS.stm. Given a source language expression which evaluates to a base value n, evaluating the result of your CPS translator using CPS.evaluate should also result in n. The file "cps-ast.sml" contains a couple of functions that you may find useful: newvar and eqvar.