CS 510 Programming Languages
Final Exam

Out: 12:00 Noon Friday, Jan. 10, 2003
Due: 12:00 Noon Friday, Jan. 17, 2002

This midterm should be the individual work of each student in the class. Please to not talk to anyone other than the professor (David Walker) about the questions on this midterm. Talking to anyone other than me about this exam prior to handing the exam in constitutes a violation of Princeton's code of academic integrity. You may consult your lecture notes, or any of the textbooks listed as required or recommended on the course Web pages. If you need clarification on a question, please e-mail me or come to see me by Monday January 13th (5 PM). I will be unavailable between Tuesday January 14 and Sunday January 19th.

A couple of helpful reminders and tips:

- Read questions completely before beginning your response.
- Always state your induction hypothesis and proof methodology clearly and explicitly before writing down the details of a proof.
- Always use the exact syntax of expressions, types, judgments, etc. that you are given in a question, or clearly define the abbreviations that you're using. When in doubt, avoid abbreviations.
- When defining a new type system, set of operational rules, etc., be sure to state the form of the new judgments in question before laying out the details of the rules (unless, of course, the question specifies the form of the judgment you should use).
- If some aspect of the question is confusing or underspecified and I am unavailable, make some reasonable assumption and write down the assumption clearly before beginning the question.
- Even if you can't come up with the correct solution for a question, write down as clear and concise an explanation of your thought process and partial work as you can to get partial credit. Do not leave a question entirely blank.
- When writing out your proofs, use plenty of space (either electronically or on paper) to make them easy to read. One of the best ways is to format in each line with one true statement (judgment) on the left and the justification for that statement on the right (in terms of earlier statements and inference rules, etc.).

Good luck!

1 Definitions and Concepts

Please answer the following questions as precisely and concisely as possible. A sentence or two will do for each.

1. What is a judgment?
2. What does a proof of type safety tell us about a language?

3. What logical notions do the following programming language concepts correspond to:
   (a) functional programs
   (b) reduction (i.e.: evaluation)
   (c) types
4. What is unification?
5. What is the difference between a synchronous operation and an asynchronous operation?

2 Closure Conversion

One of the major steps in compiling a language like ML into a language like C is *closure conversion*. Nested ML functions are allowed to contain free variables defined in outer scopes whereas C functions cannot be nested inside one another and therefore are closed (except for global variables). Closure conversion takes code involving functions with free variables and transforms it into code in which functions no longer have any free variables—the variables that were once free are passed explicitly as arguments to functions. For example, the following code fragment defines the function `f` which contains free variables `y` and `x`:

```
let val x = 1
    val y = 2
    val z = 3
    val f = fn w => x + y + w
in
    f 100
end
```

We can eliminate the free variables `y` and `x` by transforming the code and function `f` above as follows.

```
let val x = 1
    val y = 2
    val z = 3
    val f = ((fn (env,w) =>
                  let val x = #1 env
                      val y = #2 env
                  in
                      x + y + w
                  end),
             (x,y)) in
...
```

Notice that the code in the example is now closed. `(fn (env,w) => ...)` contains no free variables. Instead, it has an extra argument `env`. Inside the function, The variables `x` and `y` are extracted from the argument `env` before the function continues to compute its result. We replace the formerly open function with a closure (a pair) containing the closed code and the environment, which is itself a tuple containing `x` and `y`.

In general, to closure convert a function `fn x => e`, we follow the following steps

1. Compute the free variables of the function. Let *vars* be the list of free variables.
2. Replace the function with a pair of
   - a function that takes a pair `(env,x)` as an argument and has a transformed body `e'`
   - an environment that holds all of the formerly free variables

3. The transformed body `e'` should be constructed by projecting each variable in the list *vars* one by one from the parameter `env` and then recursively transforming the original function body.

Since we change the form of functions during this transformation, we also must change the way we apply functions. Every function call is changed as follows.

1. Extract the closed code from the closure.
2. Extract the environment from the closure.
3. Apply the closed code to a pair consisting of the environment and the original argument.

Following this recipe, we can finish off our closure conversion of the example above as follows.

```
let val x = 1
    val y = 2
    val z = 3
    val f = ((fn (env,w) =>
                  let val x = #1 env
                      val y = #2 env
                  in
                      x + y + w
                  end),
             (x,y)) in
(#1 f) (#2 f, z)     (* transformed function call *)
end
```

2.1 Untyped Closure Conversion

Answer the following questions about untyped closure conversion.

1. Closure convert the following SML code by hand, following the recipe given above. To make sure that you have done the transformation correctly, test your transformed code using SML/NJ. Notice that `twice` is a function that returns a function. When transformed it should be a closure (even though the environment is the trivial, empty one) and the code for the closure should return another closure.

   ```
   let val y = 1 in
   let val plusone = (fn x => x + y) in
   let val twice = (fn f => (fn x => f (f x))) in
   (twice plusone) 3
   ```

2. Formalize the closure conversion algorithm as a translation from (potentially open) untyped lambda calculus terms to (closed) lambda terms.

   $$\begin{array}{lrcl} \text{(terms)} & e & ::= & x \mid \mathtt{let}\, x = e_1 \,\mathtt{in}\, e_2 \mid \lambda x.e \mid e_1\, e_2 \\ & & & \mathtt{true} \mid \mathtt{false} \mid \mathtt{if}\, e_1 \,\mathtt{then}\, e_2 \,\mathtt{else}\, e_3 \end{array}$$

   Your translation judgments should be defined as simple equations on the structure of the input terms:

   $$\text{Trans}(e) = e'$$

   You may assume the existence of a function FV, which when given a lambda term e, computes the free variables of that term.

2.2 Typed Closure Conversion

Consider what happens when you attempt to translate the following if statement.

```
if true then
  let y = 3 in
  let z = 7 in
  λx . x + y + z
else
  λx . 3 + x
```

The resulting code after closure conversion is:

```
if true then
  let y = 3 in
  let z = 7 in
  ((λ(env,x) . let y = env.1 in let z = env.2 in x + y + z), (y,z))
else
  ((λ(env,x) . 3 + x), ())
```

This code is not well typed in the simply-typed lambda calculus! The data structure returned by the first branch of the if statement has type:
`((int * int) * int) -> int) * (int * int)`
whereas the second branch has type:
`((() * int) -> int) * ()`
In general, the type of the data structure depends upon the types of all of the free variables in the environment. In the first branch, the type of the environment is `(int * int)` whereas in the second branch, the type of the environment is `()`.

In this section of the exam, you will define a type directed translation that translates simply-typed lambda terms (with booleans) into lambda terms with booleans and existential types.

Simply-typed lambda calculus with booleans:

$$\begin{array}{llll} \text{(types)} & \tau & ::= & \texttt{bool} \mid \tau_1 \rightarrow \tau_2 \\ \\ \text{(terms)} & e & ::= & x \mid \texttt{let}\, x \,=\, e_1 \,\texttt{in}\, e_2 \mid \lambda x{:}\tau.e \mid e_1\, e_2 \\ & & & \texttt{true} \mid \texttt{false} \mid \texttt{if}\, e_1 \,\texttt{then}\, e_2 \,\texttt{else}\, e_3 \end{array}$$

Lambda calculus with booleans, general tuples and existentials:

$$\begin{array}{llll} \text{(types)} & \tau & ::= & \texttt{bool} \mid \tau_1 \rightarrow \tau_2 \\ & & & <\tau_1, \ldots, \tau_n> \mid \alpha \mid \exists\alpha.\tau \\ \\ \text{(terms)} & e & ::= & x \mid \texttt{let}\, x \,=\, e_1 \,\texttt{in}\, e_2 \mid \lambda x{:}\tau.e \mid e_1\, e_2 \mid \\ & & & \texttt{true} \mid \texttt{false} \mid \texttt{if}\, e_1 \,\texttt{then}\, e_2 \,\texttt{else}\, e_3 \mid \\ & & & <e_1, \ldots, e_n> \mid e.i \mid \texttt{pack}[\tau, e]\texttt{as}\exists\alpha.\tau' \mid \texttt{unpack}\,\alpha, x = e_1 \,\texttt{in}\, e_2 \end{array}$$

The typing rules and operational semantics for the language are just like the typing rules and operational semantics for MinML with similar features (the only difference is that we will work with non-recursive functions here).

1. Define the type translation for the closure conversion transformation (by induction on the structure of the simple types).

$$\texttt{T}(\tau) = \tau'$$

   Hint: Remember, when you translate *any* function with type $\tau_1 \rightarrow \tau_2$ into a closure, the closure must have the type given by $\texttt{T}(\tau_1 \rightarrow \tau_2)$. Hint: Use an existential type.

2. Define a type-directed term translation. Your term translation judgments should have the form:

$$\Gamma \vdash e : \tau \Rightarrow e'$$

   Once again, you may assume the existence of a function FV that calculates the free variables of an expression.

3. Prove that your type-directed translation preserves types. Be sure to state the theorem you are proving and your induction hypothesis precisely. State formally (but do not prove) any lemmas concerning FV that you require in your proof.

   If you are unable to prove an appropriate type-preservation theorem then try changing your term translation appropriately (or type translation). If you still can't prove the theorem then you'll get partial credit for stating a reasonable theorem and proof methodology and showing where your proof breaks down.

3 Concurrency

In this section, we will study the semantics of a concurrent programming language.[1]

Concurrent programming languages are little different than functional programming languages. One of the things that makes them different is that processes don't return values as results. A process get to work done by passing values to other processes along channels. When a process has finished its job (if indeed it ever finishes) it simply exits.

Below is a listing of the abstract syntax of the concurrent language that we will study in this question. We use the metavariable i to range over integers and c to range over CML-style channels. We use v to range over values (either booleans or channels) and we use either P or Q to range over processes.

types	τ	$::=$	$\texttt{bool} \mid \texttt{int} \mid \tau\ \texttt{chan}$
values	v	$::=$	$x \mid \texttt{true} \mid \texttt{false} \mid i \mid c$
operators	$\circ$	$::=$	$+ \mid - \mid \leq$
processes	P, Q	$::=$	$\texttt{let}\ x = v\ \texttt{in}\ P \mid \texttt{let}\ x = v_1 \circ v_2\ \texttt{in}\ P \mid \texttt{if}\ v\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \mid \texttt{exit} \mid \texttt{spawn}\ P_1; P_2 \mid \texttt{send}\ v_1\ \texttt{on}\ v_2; P \mid \texttt{recv}\ x\ \texttt{on}\ v; P \mid \texttt{new}\ x{:}\tau; P \mid \alpha \mid \texttt{rec}\ \alpha.P$

Most of the process forms above should be somewhat familiar to you from your study of CML. Here is an informal explanation of some of the commands:

- `exit` simply terminates a process.
- $\texttt{spawn}\ P_1; P_2$ spawns the new process P_1 and continues to execute P_2 concurrently with P_1.
- $\texttt{send}\ v_1\ \texttt{on}\ v_2; P$ is a synchronous operation that sends the value v_1 on the channel v_2 when there is another process that is waiting to receive on v_2.
- $\texttt{new}\ x{:}\tau; P$ generates a new channel (which is bound to the variable x) for sending and receiving values of type τ and then continues to execute P. (P may contain x.)
- $\texttt{rec}\ \alpha.P$ is a recursive process definition. To evaluate this process, substitute the entire process for the process variable α, which may appear in P.

[1] For those of you who are curious, it is related to a generic sort of concurrent language called the π-calculus. The π-calculus is to concurrent programming what the λ-calculus is to functional programming.

The operational semantics for the language is defined as a relation between abstract machine states. A machine state contains the list of channels allocated so far and the list of processes that are currently executing.

$$\text{machine states} \quad M \quad ::= \quad (c_1, \ldots, c_m; P_1, \ldots, P_n)$$

We will implicitly assume that all channels $c_1, \ldots, c_m$ in a machine state are different from another. The operational semantics is defined by the inference rules below. We'll use the notation $P[v/x]$ for our standard capture-avoiding substitution.

$\boxed{M \mapsto M'}$

$$\frac{P_i = (\texttt{let}\, x \;=\; v\, \texttt{in}\, Q)}{(c_1, \ldots, c_m; \ldots, P_{i-1}, P_i, P_{i+1}, \ldots) \mapsto (c_1, \ldots, c_m; \ldots, P_{i-1}, Q[v/x], P_{i+1}, \ldots)} \quad (\text{LET})$$

$$\frac{P_i = (\texttt{let}\, x \;=\; v_1 \,\circ\, v_2\, \texttt{in}\, Q \quad v = v_1 \,\circ\, v_2)}{(c_1, \ldots, c_m; \ldots, P_{i-1}, P_i, P_{i+1}, \ldots) \mapsto (c_1, \ldots, c_m; \ldots, P_{i-1}, Q[v/x], P_{i+1}, \ldots)} \quad (\text{OP})$$

$$\frac{P_i = (\texttt{if}\, \texttt{false}\, \texttt{then}\, Q_1\, \texttt{else}\, Q_2)}{(c_1, \ldots, c_m; \ldots, P_{i-1}, P_i, P_{i+1}, \ldots) \mapsto (c_1, \ldots, c_m; \ldots, P_{i-1}, Q_2, P_{i+1}, \ldots)} \quad (\text{IF-FALSE})$$

$$\frac{P_i = (\texttt{if}\, \texttt{true}\, \texttt{then}\, Q_1\, \texttt{else}\, Q_2)}{(c_1, \ldots, c_m; \ldots, P_{i-1}, P_i, P_{i+1}, \ldots) \mapsto (c_1, \ldots, c_m; \ldots, P_{i-1}, Q_1, P_{i+1}, \ldots)} \quad (\text{IF-TRUE})$$

$$\frac{}{(c_1, \ldots, c_m; \ldots, P_{i-1}, \texttt{exit}, P_{i+1}, \ldots) \mapsto (c_1, \ldots, c_m; \ldots, P_{i-1}, P_{i+1}, \ldots)} \quad (\text{EXIT})$$

$$\frac{P_i = (\texttt{spawn}\, Q_1; Q_2)}{(c_1, \ldots, c_m; \ldots, P_{i-1}, P_i, P_{i+1}, \ldots) \mapsto (c_1, \ldots, c_m; \ldots, P_{i-1}, Q_1, Q_2, P_{i+1}, \ldots)} \quad (\text{SPAWN})$$

$$\frac{P_i = (\texttt{send}\, v\, \texttt{on}\, c; Q_i) \quad P_j = (\texttt{recv}\, x\, \texttt{on}\, c; Q_j)}{(c_1, \ldots, c_m; \ldots, P_i, \ldots, P_j, \ldots) \mapsto (c_1, \ldots, c_m; \ldots, Q_i, \ldots, Q_j[v/x], \ldots)} \quad (\text{SYNCH-1})$$

$$\frac{P_j = (\texttt{send}\, v\, \texttt{on}\, c; Q_j) \quad P_i = (\texttt{recv}\, x\, \texttt{on}\, c; Q_i)}{(c_1, \ldots, c_m; \ldots, P_i, \ldots, P_j, \ldots) \mapsto (c_1, \ldots, c_m; \ldots, Q_i[v/x], \ldots, Q_j, \ldots)} \quad (\text{SYNCH-2})$$

$$\frac{P_i = (\texttt{new}\, x : \tau; Q) \quad c \notin \{c_1, \ldots, c_m\}}{(c_1, \ldots, c_m; \ldots, P_{i-1}, P_i, P_{i+1}, \ldots) \mapsto (c_1, \ldots, c_m, c; \ldots, P_{i-1}, Q[c/x], P_{i+1}, \ldots)} \quad (\text{NEW})$$

$$\frac{P_i = (\texttt{rec}\, \alpha.Q)}{(c_1, \ldots, c_m; \ldots, P_{i-1}, P_i, P_{i+1}, \ldots) \mapsto (c_1, \ldots, c_m; \ldots, P_{i-1}, Q[\texttt{rec}\, \alpha.Q/\alpha], P_{i+1}, \ldots)} \quad (\text{REC})$$

3.1 Programming and Typing

Answer the following questions about this concurrent language.

1. Write a process P that sends the Fibonacci sequence on the channel c. In other words, if Q is a process that repeatedly executes a receive on the channel c then it will receive all of the Fibonacci numbers when executed in parallel with P.

   Just to jog your memory, the Fibonacci sequence is

   $$0, 1, 1, 2, 3, 5, \ldots$$

   To get the ith number, one adds the $i-1$th and $i-2$th numbers.

2. Develop a simple type system for this concurrent programming language (do not include polymorphism or subtyping of any kind at this point). Specify the sort of contexts that you are using and the form of the judgments before continuing with the specific typing rules. Hint: you will probably have two different forms of judgment (not including typing rules for abstract machine states).

3. List the kinds of errors that your type system prevents from occurring.

4. Define a typing judgment for abstract machine states and give typing rules for abstract machine states.

3.2 Subtyping

In this section we will consider extending our language from above with floating point values f with type `float`. And consider the different sorts of subtyping that arise.

1. Define a declarative subtyping relation $\tau_1 \leq \tau_2$ for our concurrent language extended with the type `float`. Be sure to give the appropriate subtyping rules for channel types. As well as giving the mathematical specification of the subtyping rules for channels, give an English explanation that justifies your decision. Also, give an example (or two) that further backs up your explanation.

2. Define the corresponding algorithmic subtyping relation.

3. Prove that your algorithmic subtyping relation is sound and complete with respect to your declarative subtyping relation.

4. Consider adding two other sorts of channels to the language "input channels" with type τ `inchan` and "output channels" with type τ `outchan`. Assume new in channels and out channels are generated in pairs using the command `new` $x, y : \tau; P$ which binds x to an in channel with type τ `inchan` and binds y to an out channel with type τ `outchan`. Any values sent on y can be received on x. Are the subtyping rules for in channels and out channels the same as for regular channels? If not, extend the declarative subtyping relations that you gave above with new rules for in channels and out channels.

4 What to Hand in

Please hand in answers to all of the questions on the exam. You may e-mail answers to me electronically or you may give solutions to Tina (CS 410).

Summary of what you need to hand in:

- Section 1: Five questions
- Section 2.1: Two questions
- Section 2.2: Three questions
- Section 3.1: Four questions
- Section 3.2: Four questions

Please put your name on the exam too.