15-312 Foundations of Programming Languages

Final Examination

December 13, 2004

Name: **Sample Solution**

Andrew User ID: `fp`

- This is an open book, open notes, closed computer exam.
- Write your answer legibly in the space provided.
- There are 18 pages in this exam, including 4 worksheets.
- It consists of 5 problems worth a total of 250 points and one extra credit question worth 25 points.
- The extra credit is recorded separately, so only attempt this after you have completed all other questions.
- You have 3 hours for this exam.
- Read the questions carefully before you answer!

Problem 1	**Problem 2**	**Problem 3**	**Problem 4**	**Problem 5**	**Total**	**EC**
50	**50**	**55**	**50**	**45**	**250**	**25**

1. Continuations (50 pts)

A popular way to compile functional programs (employed in implementations of Scheme and Standard ML of New Jersey) is to translate them to *continuation-passing style*. The basic intuition is that every function is transformed to take one additional argument, its *continuation*. Instead of returning a value, the function explicitly passes the return value to its continuation argument. In this setting, continuations are implemented as ordinary functions that represent the remainder of the computation, and throwing a value to a continuation is implemented as a function call.

As a simple example, consider the successor function $\mathsf{fn}(x.\mathsf{plus}(x, \mathsf{num}(1)))$. This might be compiled to $\lambda x.\, \lambda k.\, k\,(x+1)$ which passes its result $(x+1)$ to its continuation argument k instead of returning it. Here, and in the rest of the problem we will use abstract syntax for the source of the translation and mathematical syntax for the target of the translation.

Formally, we define an inductive translation $[\![e]\!]\,k$ which translates an expression e under a continuation k. The translation of e should pass its value to k, as exemplified in the first three cases below. For every variable x in the source we have a corresponding variable $\hat{x}$ in the target. For functions and integers, the translation is defined by the following cases:

$$\begin{array}{lcl}
[\![\mathsf{num}(n)]\!]\,k & = & k\,n \\
[\![x]\!]\,k & = & k\,\hat{x} \\
[\![\mathsf{fn}(x.e)]\!]\,k & = & k\,(\lambda\hat{x}.\,\lambda k_1.\,[\![e]\!]\,k_1) \\
[\![\mathsf{apply}(e_1, e_2)]\!]\,k & = & [\![e_1]\!]\,(\lambda x_1.\,[\![e_2]\!]\,(\lambda x_2.\,x_1\,x_2\,k))
\end{array}$$

At the top-level, the C-machine starts with the empty stack "$\bullet$", also called the *initial continuation*. In the functional representation, the initial continuation is the identity function $\lambda z.\, z$ which returns it argument as the final answer of the computation. We therefore translate a given expression at the top-level under the initial continuation $\lambda z.\, z$. For example:

$$\begin{array}{l}
[\![\mathsf{apply}(\mathsf{fn}(x.x), \mathsf{num}(3))]\!]\,(\lambda z.\, z) \\
= [\![\mathsf{fn}(x.x)]\!]\,(\lambda x_1.\,[\![\mathsf{num}(3)]\!]\,(\lambda x_2.\,x_1\,x_2\,(\lambda z.\, z))) \\
= [\![\mathsf{fn}(x.x)]\!]\,(\lambda x_1.\,(\lambda x_2.\,x_1\,x_2\,(\lambda z.\, z))\,3) \\
= (\lambda x_1.\,(\lambda x_2.\,x_1\,x_2\,(\lambda z.\, z))\,3)\,(\lambda\hat{x}.\,\lambda k.\,[\![x]\!]\,k) \\
= (\lambda x_1.\,(\lambda x_2.\,x_1\,x_2\,(\lambda z.\, z))\,3)\,(\lambda\hat{x}.\,\lambda k.\,k\,\hat{x})
\end{array}$$

1.1 (10 pts) Verify, step-by-step using the small-step operational semantics and the result of the translation above, that

$$[\![\mathsf{apply}(\mathsf{fn}(x.x), \mathsf{num}(3))]\!]\,(\lambda z.\, z) \mapsto^* 3$$

$(\lambda x_1.\, (\lambda x_2.\, x_1\, x_2\, (\lambda z.\, z))\, 3)\, (\lambda \hat{x}.\, \lambda k.\, k\, \hat{x})$

$\mapsto (\lambda x_2.\, (\lambda \hat{x}.\, \lambda k.\, k\, \hat{x})\, x_2\, (\lambda z.\, z))\, 3$
$\mapsto (\lambda \hat{x}.\, \lambda k.\, k\, \hat{x})\, 3\, (\lambda z.\, z)$
$\mapsto (\lambda k.\, k\, 3)\, (\lambda z.\, z)$
$\mapsto (\lambda z.\, z)\, 3$
$\mapsto 3$

The translation to continuation-passing style changes the types of expressions. We write $[\tau]_\sigma$ for the translation of the type τ, given a final answer type σ, and $\hat{\Gamma}$ for the context that arises from replacing every declaration $x{:}\tau$ in Γ by $\hat{x}{:}[\tau]_\sigma$. Then the defining property of the type translation is

If $\Gamma \vdash e : \tau$ *and* $\Gamma' \vdash k : [\tau]_\sigma \to \sigma$ *then* $\hat{\Gamma}, \Gamma' \vdash [\![e]\!]\, k : \sigma$.

In other words, when translating $[\![e]\!]\, k$ then k must accept the value of e (after translation) and return the final answer of type σ.

1.2 (10 pts) Give the definition of $[\tau_1 \to \tau_2]_\sigma$ in terms of $[\tau_1]_\sigma$ and $[\tau_2]_\sigma$ so that the translation of functions results in well-typed terms.

$$[\tau_1 \to \tau_2]_\sigma \quad = [\tau_1]_\sigma \to ([\tau_2]_\sigma \to \sigma) \to \sigma$$

1.3 (15 pts) Extend the translation to pairs by completing the following table. You may use pairs in the target.

$$
\boxed{
\begin{aligned}
[\tau_1 \times \tau_2]_\sigma &= [\tau_1]_\sigma \times [\tau_2]_\sigma \\
\llbracket \mathsf{pair}(e_1, e_2) \rrbracket\, k &= \llbracket e_1 \rrbracket\, (\lambda x_1.\, \llbracket e_2 \rrbracket\, (\lambda x_2.\, k\, (\mathsf{pair}(x_1, x_2)))) \\
\llbracket \mathsf{fst}(e) \rrbracket\, k &= \llbracket e \rrbracket\, (\lambda x.\, k\, (\mathsf{fst}(x))) \\
\llbracket \mathsf{snd}(e) \rrbracket\, k &= \llbracket e \rrbracket\, (\lambda x.\, k\, (\mathsf{snd}(x)))
\end{aligned}
}
$$

1.4 (15 pts) Translation to continuation-passing style makes it very easy to implement callcc and throw in the source without using callcc or throw in the target, since continuations are represented as ordinary functions. For reference, here are the typing rules for callcc and throw.

$$
\frac{\Gamma, x{:}\tau\ \mathsf{cont} \vdash e : \tau}{\Gamma \vdash \mathsf{callcc}(x.e) : \tau}
\qquad\qquad
\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau\ \mathsf{cont}}{\Gamma \vdash \mathsf{throw}(e_1, e_2) : \tau'}
$$

Complete the following definitions. [Hint: use the types as your guide.]

$$
\boxed{
\begin{aligned}
[\tau\ \mathsf{cont}]_\sigma &= [\tau]_\sigma \to \sigma \\
\llbracket \mathsf{callcc}(x.e) \rrbracket\, k &= \{k/\hat{x}\}(\llbracket e \rrbracket\, k) \\
\llbracket \mathsf{throw}(e_1, e_2) \rrbracket\, k &= \llbracket e_1 \rrbracket\, (\lambda x_1.\, \llbracket e_2 \rrbracket\, (\lambda x_2.\, x_2\, x_1))
\end{aligned}
}
$$

2. Object-Oriented Programming (50 pts)

In this question we will examine the correct use of object-oriented constructs. Suppose we are writing software to handle the routing and shipping of packages. We have various places that produce packages, and various parties that want consume them. Part of the declarations in the library shown below:

```
class Producer of { ... }
class Consumer of { ... }
class Package of { ... }
class RemoteProducer extends Producer of { ... }
class FragilePackage extends Package of { ... }
class RemoteConsumer extends Consumer of { ... }

method deliver(Producer, Package, Consumer) : unit
```

In this library, we have classes to account for all of the producers, packages, and consumers, and a method which takes one of each to take care of all behavior for one delivery. Far-away production and consumption sites, and fragile packages may require special treatment, and so we subclass each of the three base classes.

Suppose a few of our coworkers in this fictional shipping company come to us and tell us they have finished implementing the method `deliver`. The code they have written looks like the following.

```
extend deliver(p : RemoteProducer, k : Package, c : RemoteConsumer) = ...
extend deliver(p : RemoteProducer, k : Package, c : Consumer) = ...
extend deliver(p : Producer, k : FragilePackage, c : Consumer) = ...
```

2.1 (10 pts) Given (only) these declarations, will the typechecker (using only the global checking algorithm described in lecture—don't worry about the more efficient local checking techniques) report that the program is vulnerable to a run-time '*message not found*' error? (Remember that this is analogous to ML's '*nonexhaustive match exception*') If so, give an example of a method call that will result in this error. You can write simply {`Package: ...`}, for example, to indicate an object of class Package, and similarly with the other five classes.

> Yes. A method call that triggers this error is
> `call deliver({Producer:...},{Package:...},{Consumer:...})`

2.2 (15 pts) Given (only) these declarations, will the typechecker report that the program is vulnerable to a run-time '*message ambiguous*' error? If so, give an example of a method call that will result in this error.

Yes. A method call that triggers this error is
```
call deliver({RemoteProducer:...},{FragilePackage:...},
{Consumer:...})
```

2.3 (10 pts) If we add a case

```
extend deliver(p : Producer, k : Package, c : Consumer) = ...
```

do one or both of the above answers change? Explain.

The '*message not found*' error is no longer possible, but the extended program is still vulnerable to a '*message ambiguous*' error, on the same method call as before.

Now, consider a similar library implemented in Java. The `deliver` function has become a method of class `Producer`, and a default implementation is given.

```
class Producer {
  void deliver(Package k, Consumer c) {
   ...
  }
class Consumer  { ... }
class Package   { ... }
}
```

A programmer who learned to program in EML might write the following Java code:

```
class FragilePackage extends Package  { ... }
class RemoteConsumer extends Consumer  { ... }
class RemoteProducer extends Producer {
   void deliver(Package p, Consumer c) { ...          /* (1) default */ }
   void deliver(FragilePackage p, Consumer c) { ...   /* (2) fragile */ }
   void deliver(Package p, RemoteConsumer c) { ...    /* (3) remote */  }
   void deliver(FragilePackage p, RemoteConsumer c) { /* (4) fragile and
                                                             remote */ }
}
```

2.4 (15 pts) If our client writes the following code

```
Producer p = new RemoteProducer(...);
Package k = new FragilePackage(...);
Consumer c = new RemoteConsumer(...);
p.deliver(k,c);
```

Which of (1),(2),(3),(4) gets executed? Is this the right behavior? If not, explain how the implementation of `deliver` can be fixed.

Java resolves statically overloaded functions based on the types of the arguments. In this case, the `deliver` function is being called with arguments of (static) type `Package` and `Consumer`. Therefore (1) will be invoked.
This is almost certainly not what the programmer intended. There is special-case behavior for `RemoteConsumer`s and `FragilePackage`s that is not being called. To get the right behavior, we ought to have a single method declaration in `RemoteProducer` that uses `instanceof` to distinguish between the possibilities for its arguments.

```
void deliver(Package k, Consumer c) {
  bool remote = c instanceof RemoteConsumer;
  bool fragile = k instanceof FragilePackage;
  if (!remote) {
   if (!fragile) {
    /* (1) */
   }
   else {
    /* (2) */
   }
  }
  else {
   if (!fragile) {
    /* (3) */
   }
   else {
    /* (4) */
   }
  }
}
```

A more complex solution that is cleaner in some ways is to use the Visitor design pattern. This solution is beyond the scope of this exam, however.

3. Monads and Subtyping (55 pts)

Monads are an important device to isolate effects in a pure functional language. For reference, here are the generic constructs for monads, independent of any particular notion of effect.

Types	τ	$::=$	$\cdots \mid \bigcirc\tau$
Pure Expressions	e	$::=$	$\cdots \mid \mathsf{comp}(m)$
Monadic Expressions	m	$::=$	$e \mid \mathsf{letcomp}(e, x.m)$

In concrete syntax we may write $\mathsf{let\,comp}\ x = e\ \mathsf{in}\ m\ \mathsf{end}$ for $\mathsf{letcomp}(e, x.\,m)$. Recall the following typing rules, which include the new judgment $\Gamma \vdash m \div \tau$ for monadic expressions

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \div \tau} \qquad \frac{\Gamma \vdash m \div \tau}{\Gamma \vdash \mathsf{comp}(m) : \bigcirc\tau} \qquad \frac{\Gamma \vdash e : \bigcirc\tau \quad \Gamma, x{:}\tau \vdash m \div \sigma}{\Gamma \vdash \mathsf{letcomp}(e, x.\,m) \div \sigma}$$

and recall that $\mathsf{comp}(m)$ is a value.

In this problem we explore if subtyping could be used to reduce the syntactic overhead of programming with monads. For this set of questions, we adopt the coercion interpretation of subtyping. For each of the proposed rules, state whether it satisfies the *Fundamental Principle of Subtyping* discussed in lecture or not. If it does, write out the omitted coercion f.

3.1 (10 pts)

$$\frac{}{f : \bigcirc\tau \leq \tau}$$

Satisfies FPS? (circle one) Yes [✓ No]

$\boxed{\text{There is no general way to coerce a computation to a value.}}$

3.2 (10 pts)

$$\frac{}{f : \tau \leq \bigcirc\tau}$$

Satisfies FPS? (circle one) [✓ Yes] No

$\boxed{\text{Coercion } f = \lambda x.\mathsf{comp}(x)}$

3.3 (15 pts)

$$\frac{}{f : \bigcirc\bigcirc\tau \leq \bigcirc\tau}$$

Satisfies FPS? (circle one) [✓ Yes] No

Coercion $f = \lambda x.\mathsf{comp}(\mathsf{let\,comp}\ y = x\ \mathsf{in}\ \mathsf{let\,comp}\ z = y\ \mathsf{in}\ z\ \mathsf{end}\ \mathsf{end})$

3.4 (15 pts)

$$\frac{g : \tau \leq \sigma}{f : \bigcirc\tau \leq \bigcirc\sigma}$$

Satisfies FPS? (circle one) [✓ Yes] No

Coercion $f = \lambda x.\mathsf{comp}(\mathsf{let\,comp}\ y = x\ \mathsf{in}\ g\,y\ \mathsf{end})$

3.5 (5 pts) Would your answers to the questions above change for a particular monad such as, for example, the store monad? Explain very briefly.

No, since the coercions above are independent of any particular effects.

4. Program Equivalence (50 pts)

Our definition of equivalence for functional programs in a call-by-value language distinguishes between equivalence $e \cong e' : \tau$ on expressions and $v \simeq v' : \tau$ on values. Recall the following clauses in their definition.

$$
\begin{array}{lll}
e \cong e' : \tau & \text{iff} & \text{either } e \text{ diverges and } e' \text{ diverges} \\
 & & \text{or } e \mapsto^* v \text{ and } e' \mapsto^* v' \text{ with } v \simeq v' : \tau \\[1em]
v \simeq v' : \mathsf{int} & \text{iff} & v = v' = n \text{ for an integer } n. \\
v \simeq v' : \tau_1 \to \tau_2 & \text{iff} & \text{for all } v_1 \simeq v_1' : \tau_1 \text{ we have } v\, v_1 \cong v'\, v_1' : \tau_2
\end{array}
$$

4.1 (10 pts) Our definition is not appropriate for a call-by-name language. Give two functions which would be considered equal by the definition above, yet behave differently in a call-by-name language. You may use functions, integers (including primitive operations as needed), and recursion.

We have

$$\lambda x.0 \simeq \lambda x.x - x : \mathsf{int} \to \mathsf{int}$$

However, in a call-by-name language the first will return 0 when applied to a non-terminating expression of type int, while the second one will diverge.

4.2 (10 pts) Returning to the case of a call-by-value language, add an appropriate clause for sums, $\tau_1 + \tau_2$.

$$
\begin{array}{lll}
v \simeq v' : \tau_1 + \tau_2 & \text{iff} & v = \mathsf{inl}(v_1) \text{ and } v' = \mathsf{inl}(v_1') \text{ with } v_1 \simeq v_1' : \tau_1, \text{or} \\
 & & v = \mathsf{inr}(v_2) \text{ and } v' = \mathsf{inr}(v_2') \text{ with } v_2 \simeq v_2' : \tau_2
\end{array}
$$

In the remainder of this question we consider the *output monad* with one additional form of monadic expression, $\mathsf{write}(e)$, which evaluates the expression e to an integer, writes the result to the output stream and also returns it.

$$\frac{\Gamma \vdash e : \mathsf{int}}{\Gamma \vdash \mathsf{write}(e) \div \mathsf{int}}$$

The rules in the small-step operational semantics for monadic expressions than have the form $\langle s, m\rangle \mapsto \langle s', m'\rangle$ where s and s' represent the output stream as a potential infinite sequence of integers.

4.3 (10 pts) Give the transition rule(s) for the new write construct in a small-step operational semantics. You do not need to repeat the generic monad rules.

$$\frac{e \mapsto e'}{\langle s, \mathsf{write}(e)\rangle \mapsto \langle s, \mathsf{write}(e')\rangle} \qquad \frac{}{\langle s, \mathsf{write}(\mathsf{int}(n))\rangle \mapsto \langle s \cdot n, \mathsf{int}(n)\rangle}$$

4.4 (20 pts) We extend the definition of observational equivalence to include monadic expressions with a new judgment, $\langle s, m\rangle \cong \langle s', m'\rangle \div \tau$. Complete the following definitions, assuming that we can observe the output of a computation only when it terminates.

$\langle s, m\rangle \cong \langle s', m'\rangle \div \tau$ iff $\langle s, m\rangle$ and $\langle s', m'\rangle$ both diverge,
or $\langle s, m\rangle \mapsto^* \langle s_O, v\rangle$ and $\langle s', m'\rangle \mapsto^* \langle s_O, v'\rangle$
with $v \simeq v' : \tau$ for some s_O, v, v'.

$v \simeq v' : \bigcirc\tau$ iff $v = \mathsf{comp}(m)$ and $v' = \mathsf{comp}(m')$ and
$\langle \cdot, m\rangle \simeq \langle \cdot, m'\rangle : \tau$

4.5 (25 pts **extra credit**) Revise your definition in question 4.4 under the assumption that we can observe the output of non-termination computations as they proceed.

answer omitted

5. Bisimulation (45 pts)

Consider the following three process definitions.

$$\begin{array}{rcll} P & = & a.P + b.a.P + a.a.P & \\ Q & = & a.Q_1 + \tau.Q_1 \qquad \text{where} \qquad & Q_1 = b.a.Q + \tau.Q \\ R & = & a.R + b.a.R & \\ S & = & a.S \mid b.a.S & \end{array}$$

In each of the following cases indicate whether a weak bisimulation exists or not. If it does, give the bisimulation, making additional process definitions if necessary. [Hint: it may be helpful to draw some labeled transition graphs.]

5.1 (15 pts) $P \approx Q$? Circle one: Yes [✓ No]

Not weakly bisimilar, because P can get into a state where only a is possible, while Q and Q_1 can both exhibit b.

5.2 (15 pts) $Q \approx R$? Circle one: [✓ Yes] No

If yes, show the weak bisimulation:

$Q \approx R$, $Q_1 \approx R$, and $a.Q \approx a.R$ is a weak bisimulation.

5.3 (15 pts) $R \approx S$? Circle one: Yes [✓ No]

S can exhibit the sequence a, b, b while R cannot.