CHALMERS TEKNISKA HÖGSKOLA
Dept. of Computer Science and Engineering
Jean-Philippe Bernardy

Afternoon, March 21st (Location:V).
Programming Paradigms
DAT121 / DIT331(GU)

Exam in Programming Paradigms

Afternoon, March 21st (Location:V).

Examiner: Jean-Philippe Bernardy
Permitted aids: Pen and blank paper.

There are 5 questions, each worth 12 points. The total sum is 60 points.

Some questions come with *remarks*: you must take those into account. Some questions come with *hints*: even though they are meant to help you, you may ignore those.

You will be asked to write programs in various paradigms. Choose the language appropriately in each case, and indicate which you choose at the beginning of your answer.

Paradigm	Acceptable language
Imperative	C or (an OO language where you refrain to use Objects)
Object oriented	C++ or Java
Functional	Haskell, ML
Concurrent	Erlang, Concurrent-Haskell
Logic	Prolog, Curry

You may also use pseudo-code resembling an actual language in the relevant list. In that case, make sure your code can only be interpreted in the way you intent (the responsibility lies with you). In particular, in the case of functional/logic languages, omitting necessary parentheses is NOT acceptable: `a b c` means `(a b) c` and is not acceptable pseudo-code for `a (b c)`.

Chalmers: 24 points is required to pass (grade 3), 36 points is required for grade 4, and 48 points is required for grade 5.

GU: 24 points is required to pass (grade G) and 42 points is required for grade VG.

1 Objects, Records, Method pointers

Consider the following class hierarchy written in a Java-like language:

```
class A {
  float x;
  float y;
};

class B {
  int n;
  // this method is overridable (the default in Java)
  // in C++ it would be marked 'virtual'
  void f(A p) {
    n = n+n;
  };

};

class C extends B {
  int m;
  // the following method overrides f from class B
  void f(A p) {
    n = n+n;
    m = m+n;
  }
};

static void test(A a, B b) {
  b.f(a);
}
```

- Translate the above classes (A,B,C) to records with explicit method pointers. Remark: remember to write the code which initialises the explicit method pointers. Assume that the arguments are passed by reference. **6 points**
- Translate the static method (function) test. Assume that the arguments are passed by reference. **3 points**
- Translate the static method (function) test. Assume that the argument b is passed by value (all the others are still passed by reference.) **3 points**

Answer:

-
```
struct A {
  float x = 0
  float y = 0
```

```
}
struct B {
  int n = 0
  void* f(B* this, A* p) = B_f;
}
struct C {
  int n = 0
  void* f(C* this, A* p) = C_f;
  int m;
}
```

- ```
  void test(A* a, B* b) {
    b->f(b,a);
  }
  ```

- ```
  void test(A* a, B b) {
    B_f(b,a);
  }
  ```

2 Continuations and Recursion

Consider the following function, which computes the fibbonaci number of its argument.

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

1. How many non-tail calls to fib does the above function contain? **2 points**
2. True or false: a tail-call needs stack space to be implemented. Justify your answer. **2 points**
3. Translate the above function to use explicit continuations.
   Remark: Assume an ambient type of effects called Effect.
   (a) Write the type of the translated fib function. **2 points**
   (b) Write the body of the translated fib function. **4 points**
4. How many non-tail calls to fib does the (correctly) translated function contain? **2 points**

Answer:

1. 2
2. False. None of the data saved on the stack will be used after a tail call returns.
3. ```
   fib :: Int -> (Int -> eff) -> eff
   fib 0 k = k 1
   fib 1 k = k 1
   fib n k = fib (n-1) $ \x ->
             fib (n-2) $ \y ->
             k (x+y)
   ```
4. 0
   ```

3 Closures

Consider the following piece of Haskell code.

```haskell
data Point = Point Float Float
type Region = Point -> Bool

(.+.) :: Point -> Point -> Point
Point x1 y1 .+. Point x2 y2 = Point (x1 + x2) (y1 + y2)

opposite :: Point -> Point
opposite (Point x y) = Point (negate x) (negate y)

(.-.) :: Point -> Point -> Point
p1 .-. p2 = p1 .+. opposite p2

norm2 :: Point -> Float
norm2 (Point x y) = x*x + y*y

outside :: Region -> Region
outside r = \p -> not (r p)

intersect :: Region -> Region -> Region
intersect r1 r2 p = r1 p && r2 p

withinRange :: Float -> Point -> Region
withinRange range p1 p2 = norm2 (p1 .-. p2) <= range * range
```

Transform the above code to use explicit closures for the `Region` type.

- Create a data type containing a constructor for each operation constructing a `Region` **3 points**
- Give the type of a function `apply` which converts each constructor of the `Region` data type into a function (`Point -> Bool`). **3 points**
- Give its definition **3 points**
- Give the new code for each operation, by referencing to the constructors of the new `Region` data type **3 points**

Answer:

```haskell
data Region = Outside Region
            | Intersect Region Region
            | Within Float Point
```

```haskell
apply :: Region -> Point -> Bool
apply (Outside r) p       = not (apply r p)
apply (Intersect r1 r2) p = apply r1 p && apply r2 p
apply (Within range p1) p2 = norm2 (p1 .-. p2) <= range * range

outside :: Region -> Region
outside r = Outside r

intersect :: Region -> Region -> Region
intersect r1 r2 = Intersect r1 r2

withinRange :: Float -> Point -> Region
withinRange range p1 p2 = Within range p1 p2
```

4 State-Management Process

Assume a language without pointers nor variables, but support for concurrency. In particular, assume primitives for creating processes and channels, and primitives for reading and writing to to channels. For example C++-style:

```
Chan<A> newChan();
A readChan(Chan<A> c);
void writeChan(Chan<A> c,A x);
void forkProcess(*void());
```

or Haskell-style:

```
newChan :: IO (Chan a)
readChan :: Chan a -> IO a
writeChan :: Chan a -> a -> IO ()
forkProcess :: IO () -> IO ()
```

Your task is to simulate references to mutable variables using the above primitives. This can be done be using a process that manages the variable state.

1. Define the representation for a variable of type "reference to Integer". Name this type `Reference`. **3 points**
2. Write the code for the process that manages the variable state. **4 points**
3. Write the code for primitives to create, read and write references. Their type should be: **3 points**

   ```
   Reference newRef();
   int readRef(Reference);
   void writeRef(Reference, int);
   ```

   or Haskell-style:

   ```
   newRef :: IO Reference
   readRef :: Reference -> IO Int
   writeRef :: Reference -> Int -> IO ()
   ```

4. Explain what happens during a readRef operation. In particular:
   (a) How many channel(s) are used? **1 points**
   (b) For each of the channel(s), how many message(s) are sent? **1 points**

Hint: The channels can transmit any type of information, including references to channels.

Remark: You can use either a functional or imperative language to write your answer, however you may not use any global variable nor primitive reference types in it.

Answer:

```haskell
data Command a = Get (Chan a) | Set a
type Variable a = Chan (Command a)

handler :: Variable a -> a -> IO ()
handler v a = do
  command <- readChan v
  case command of
    Set a' -> handler v a'
    Get c -> do
      writeChan c a
      handler v a

newVariable :: a -> IO (Variable a)
newVariable a = do
  c <- newChan
  forkIO (handler c a)
  return c

get :: Variable a -> IO a
get v = do
  c <- newChan
  writeChan v (Get c)
  readChan c

set :: Variable a -> a -> IO ()
set v a = do
  writeChan v (Set a)
```

5 Unification

```haskell
    both f (x,y) = (f x, f y)
    data Term = Con String [Term] -- the terms are the arguments to the constant
              | Var Int -- metavariable
      deriving Eq
(1) unify :: [(Term,Term)] -> Substitution -> Maybe Substitution
(2) unify [] s = Just s -- Base case
(3) unify ((t,t'):ts) s | t == t' = unify ts s
(4) unify ((Con f as,Con g bs):ts) s
(5)   | f == g && length as == length bs = unify (zip as bs ++ ts) s
(6)   | otherwise = Nothing -- Clash
(7) unify ((Var x,t):ts) s = unify (map (both (applySubst (x ==> t))) ts) (s +> (x,t))
(8) unify ((t, Var x):ts) s = unify ((Var x, t):ts) s -- Re-orient
```

```haskell
-------------------------------------
-- Helper functions: Substitution

-- | Add an "assignment" to a substitution
(+>) :: Substitution -> Substitution -> Substitution
bigS +> smallS = M.union (applySubst smallS `fmap` bigS) smallS

-- | Single substitution
(==>) :: String -> Term -> Substitution
x ==> t = M.singleton x t

-- | Apply a substitution to a term
applySubst :: Substitution -> Term -> Term
applySubst s (Var x) = case M.lookup x s of
  Nothing -> Var x
  Just t -> t
applySubst s (Con cname args) = Con cname (map (applySubst s) args)
```

The above algorithm for unification does not perform the occurs test.

1. Give an example of a unification constraints which fails if the occurs test is enabled **3 points**
2. Give and explain a solution (substitution) of your example, assuming the occurs test is disabled **3 points**
3. The occurs test can be added by adding a single line in the above algorithm. Between which lines should the test be added? **3 points**
4. Write the line in question. (You may assume standard Term-manipulation functions.) **3 points**

Answer:

1. Cons 0 x == x
2. Cons 0 (Cons 0 ...)
3. 6/7
4.
```haskell
      unify ((Var x,t):ts) s | x `occursIn` t = Nothing
```