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

Afternoon, August 29th, 2014 (Location:V).
Programming Paradigms
DAT121 / DIT331(GU)

Exam in Programming Paradigms

Afternoon, August 29th, 2014 (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 Recursion and Loops

Consider the following C code:

```
int s(int n) {
  int t = 0;
  while (n != 1) {
    t = t+1;
    if (n & 1 == 0)
      n = n/2;
    else
      n = 3*n + 1;
  }
  return t;
}
```

Convert the above to a functional program, which does not use state (modifiable variables).
Remark: you should use recursion to do so.
Hint: the operator `&` translates to `.&.` in Haskell.

Answer:

```
s 1 t = t
s n t = if n .&. 1
          then s (n/2)   (t+1)
          else s (3*n+1) (t+1)
```

2 Algebraic Types and Objects

Consider the following Haskell code:

```
data Expr = Const Int | Plus Expr Expr | Mult Expr Expr

eval (Const x) = x
eval (Plus x y) = eval x + eval y
eval (Mult x y) = eval x * eval y
```

Translate the code to an object oriented language.

- Write an interface corresponding to the declaration of Expr. It should contain a method corresponding to `eval` **4 points**
- Write a class for for the `Const` case. **4 points**
- Write a class for for the `Plus` case. **4 points**

Hint: you may leave the `Mult` case out, as it is the same as `Plus`.

Answer:

```
interface Expr { int eval(); }

class Const implements Expr { int value; int eval() {return
    value;} }

class Plus implements Expr { Expr l; Expr r; int eval() {return
    r.eval() + r.eval();}
}
```

3 Pattern matching and Higher-Order Abstractions

Consider the following code:

```
foobar :: [Int] -> Int
foobar []     = 0
foobar (x:xs)
  | x > 20     = (5*x-20) + foobar xs
  | otherwise = foobar xs

sum [] = 0
sum (x:xs) = x + sum xs

map f [] = []
map f (x:xs) = (f x) : (map f xs)

filter f [] = []
filter f (x:xs) = if f x then x:(filter f xs) else filter f xs

fold k f [] = k
fold k f (x:xs) = f x (fold k f xs)
```

- Express the function `foobar` in terms of `sum`, `map` and `filter`. **6 points**
- Express the function `foobar` in terms of `fold` ONLY. **6 points**

Remark: In particular you may NOT use recursion (directly) in your definitions.

Answer:

```
  foobar = sum . map (\x -> 7*x+2) . filter (>3)
  foobar = fold 0 f where
     f x acc = if x > 3 then 7*x + 2 + acc else acc
```

4 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
```

5 Functions to Relations

Consider the following Haskell code:

```
data Nat = Zero | Suc Nat

fib :: Nat -> Nat
fib Zero = Zero
fib (Suc Zero) = (Suc Zero)
fib (Suc (Suc n)) = fib (Suc n) + fib n

(+) :: Nat -> Nat -> Nat
Zero + n = n
Suc n + m = Suc (n + m)
```

Translate the function `fib` to a relation `fibo`, such that

```
fibo x y      is equivalent to      fib x == y
plus x y z    is equivalent to      x + y == z
```

1. Write the type of the relations `fibo` and `plus`. **6 points**
2. Write their code. **6 points**

Remark: you cannot use any other helper function in your answer, only constructors and relations. The types should be written in Curry or Haskell-like syntax.

Answer:

```
fibo :: Nat -> Nat -> Success
fibo Zero Zero = success
fibo (Suc Zero) (Suc Zero) = success
fibo (Suc (Suc n)) m = fibo (Suc n) a &
                       fibo n b &
                       plus a b m
  where a b free

plus :: Nat -> Nat -> Nat -> Success
plus Zero m n = m =:= n
plus (Suc m) n (Suc p) = plus m n p
```