CSEP505 Programming Languages, Autumn 2016, Final Exam
December, 2016

Programming Languages for a World of Change

Rules:

- See `http://courses.cs.washington.edu/courses/csep505/16au/exam_info.html`.
- This is a take-home exam to be completed on your own.
- There are a total of **125 points** spread **unevenly** among **10 questions**, most with subparts.

Advice:

- Read questions carefully. Understand a question before you start writing.
- Write down thoughts and intermediate steps so you can get partial credit. But clearly indicate what is your final answer.
- The questions are not necessarily in order of difficulty. You may wish to skip around. Make sure you get to all the questions.

Preamble (there is no question on this page):

The problems on this exam are all related in some way to U.S. coins — quarters, dimes, nickels, and pennies. Assume no other coins exist (until the last problem as described there). The "money value" of a collection of coins is the sum of the cents of all the coins with this "domain knowledge" that you surely already know:

coin name	coin value in cents
penny	1
nickel	5
dime	10
quarter	25

This use of the phrase "money value" is *not* the same as the notion of "value" in our study of programming languages. We will make this clear as needed in the questions that follow.

Most problems refer to either `exam.ml` or `exam.hs`, which you should look at to understand the question and edit to provide your answer. As with our homeworks, we provide `exam.fs` as an alternative to `exam.ml` but the differences are very minor.

1. (**8** points) (OCaml Warmup)

In the space indicated in `exam.ml`, implement a function `replace_pennies` as follows:

- It should have type `money -> money`
  (which is the same thing as `int * int * int * int -> int * int * int * int`).
- The result should have zero pennies.
- The result should have a money value less than or equal to the argument's money value but otherwise as large as possible.
- The result should have as few total coins as possible provided that the number of quarters does not decrease, the number of dimes does not decrease, and the number of nickels does not decrease.

For example, `replace_pennies (2,0,3,43) = (3,1,4,0)`.

Hints:

- In OCaml, the mod operator is `mod`. In F#, it is `%`.
- The sample solution is shorter than the description of it above.

Solution:

```
let replace_pennies (q,d,n,p) =
  let (q1,p) = p / 25, p mod 25 in
  let (d1,p) = p / 10, p mod 10 in
  let (n1,p) = p /  5, p mod  5 in
  (q+q1,d+d1,n+n1,0)
```

2. (**15** points) (Large-Step Interpreter)

exam.ml defines a type coin_exp for an expression language with various operations over values of type money. Part of interp_large_coin_exp of type (string * money) list -> coin_exp -> money is given to you. Complete this function to meet this description:

- A MoneyConst expression evaluates immediately to its money value. We disallow negative numbers of coins. *This case is given to you.*
- As in IMP in class, we have variables that we look up in the heap. Using an undefined variable raises an InterpFailure exception. (Not shown are statements that would create and assign to such variables.) *This case is given to you.*
- A CombineMoney expression evaluates its two subexpressions and produces a money value that has exactly all the coins produced by the two subexpressions (e.g., the number of dimes is the sum of the dimes produced by the two subexpressions).
- A RemoveCoin expression evaluates its subexpression and then produces a result that has exactly one less coin (the coin indicated by the second argument to RemoveCoin). However, if the subexpression produces money that already has 0 of the coin-to-be-removed, an InterpFailure exception should be raised.
- A HalfValue expression evaluates its subexpression then produces a result that has a subset of the coins produced by the subexpression such that the money value of the subset is half as much. If this is impossible, an InterpFailure exception should be raised. *This case is given to you.*
- A ReplacePennies expression evaluates its subexpression then produces a result that replaces all pennies as in your solution to Problem 1. No exception can occur unless it occurs in the evaluation of the subexpression.

Solution:

```
let rec interp_large_coin_exp heap exp =
  match exp with
    MoneyConst (q,d,n,p) -> if q<0 || d<0 || n<0 || p<0
                            then raise InterpFailure
                            else (q,d,n,p)
  | Var s -> lookup heap s
  | CombineMoney (e1,e2) ->
    let (q1,d1,n1,p1) = interp_large_coin_exp heap e1 in
    let (q2,d2,n2,p2) = interp_large_coin_exp heap e2 in
    (q1+q2,d1+d2,n1+n2,p1+p2)
  | RemoveCoin (e,c) ->
    let (q,d,n,p) = interp_large_coin_exp heap e in
    (match c with
      Quarter -> if q > 0 then (q-1,d,n,p) else raise InterpFailure
    | Dime    -> if d > 0 then (q,d-1,n,p) else raise InterpFailure
    | Nickel  -> if n > 0 then (q,d,n-1,p) else raise InterpFailure
    | Penny   -> if p > 0 then (q,d,n,p-1) else raise InterpFailure)
  | HalfValue e ->
    let m = interp_large_coin_exp heap e in
    let v = value_of_money m in
    if v mod 2 = 1
    then raise InterpFailure
    else (match split_money (v / 2) m with
              Some (m1,m2) -> m1
            | None -> raise InterpFailure)
  | ReplacePennies e ->
    replace_pennies (interp_large_coin_exp heap e)
```

3. (**12** points) (Higher-Order Functions and CPS)

In `exam.ml`, the function `all_coins_tree` is provided to you.

(a) Implement `penniless` using a partial application of `all_coins_tree`. `penniless` should have type `money_tree -> bool` and return `true` if and only if its argument contains no pennies.

(b) Implement `all_coins_tree_cps` of type `(coin -> bool) -> money_tree -> (bool -> bool) -> bool` by converting `all_coins_tree` to continuation-passing style. It is okay if `all_coins_tree_cps` "processes tree elements" in a different order than `all_coins_tree`.

(c) Implement `penniless2` to have the same type and functionality as `penniless` but implement it by using `all_coins_tree_cps` (and not with partial application).

Solution:

```
let rec all_coins_tree f t =
  match t with
    Leaf c -> f c
  | Node (c,t1,t2) -> f c && all_coins_tree f t1 && all_coins_tree f t2

let penniless = all_coins_tree (fun c -> c <> Penny) (* pattern-matching fine *)

let rec all_coins_tree_cps f t k =
  match t with
    Leaf c -> k (f c)
  | Node (c,t1,t2) ->
    all_coins_tree_cps f t1
    (fun b1 -> all_coins_tree_cps f t2
      (fun b2 -> k (b1 && b2 && f c))

let penniless2 t = all_coins_tree_cps (fun c -> c <> Penny) t (fun x -> x)
```

4. (**18** points) (Formal Semantics)

Here is a *formal semantics* for our coin-expression language. Each inference rule has the form $H; e \Downarrow (q, d, n, p)$ where H is a heap, e is an expression and q, d, n, p are all numbers representing, as in OCaml, the number of quarters, dimes, nickels, and pennies. Assume the Variable rule is correct even though we leave undefined the exact meaning of $H(x)$. Even so, some of the rules are *not what we intend* or otherwise have *differences* from your interpreter written in OCaml.

$$\textsc{Constant}\quad \frac{q \geq 0 \qquad d \geq 0 \qquad n \geq 0 \qquad p \geq 0}{H; \textsf{MoneyConst}\ (q, d, n, p) \Downarrow (q, d, n, p)}$$

$$\textsc{Variable}\quad \frac{H(x) = (q, d, n, p)}{H; \textsf{Var}\ x \Downarrow (q, d, n, p)}$$

$$\textsc{Combine}\quad \frac{H; e_1 \Downarrow (q, d, n, p) \qquad H; e_2 \Downarrow (q, d, n, p)}{H; \textsf{CombineMoney}(e_1, e_2) \Downarrow (q + q, d + d, n + n, p + p)}$$

$$\textsc{Half}\quad \frac{H; e \Downarrow (q, d, n, p) \qquad 25q + 10d + 5n + p = 2(25q' + 10d' + 5n' + p')}{H; \textsf{HalfValue}\ e \Downarrow (q', d', n', p')}$$

$$\textsc{RemoveQ}\quad \frac{H; e \Downarrow (q, d, n, p) \qquad q > 0}{H; \textsf{RemoveCoin}(e, \textsf{Quarter}) \Downarrow (q - 1, d, n, p)}$$

$$\textsc{RemoveD}\quad \frac{H; e \Downarrow (q, d, n, p) \qquad d > 0}{H; \textsf{RemoveCoin}(e, \textsf{Dime}) \Downarrow (q, d - 1, n, p)}$$

$$\textsc{RemoveN}\quad \frac{H; e \Downarrow (q, d, n, p) \qquad n > 0}{H; \textsf{RemoveCoin}(e, \textsf{Nickel}) \Downarrow (q, d, n - 1, p)}$$

$$\textsc{RemoveP}\quad \frac{H; e \Downarrow (q, d, n, p) \qquad p > 0}{H; \textsf{RemoveCoin}(e, \textsf{Penny}) \Downarrow (q, d, n, p - 1)}$$

$$\textsc{Replace}\quad \frac{H; e \Downarrow (q, d, n, p) \qquad 5n' + x = p \qquad 0 \leq x \leq 4}{H; \textsf{ReplacePennies}\ e \Downarrow (q, d, n + n', 0)}$$

Give your answers to this problem in a text document of your choice. You can also do it on paper and take a clear photograph of your answers if you prefer.

(a) One of the rules produces *different results* — it has the same "failure modes" as your interpreter from Problem 2 and always produces one answer, but it does not always produce the same answer. Explain in roughly 1 English sentence which rule and why it is different. Then give *two* example expressions: one where the answers are the *same* here and in your interpreter and one where the answers are *different*.

(b) One of the rules produces *fewer results* — when it gives an answer, that answer agrees with your interpreter, but it gives answers in fewer situations. Explain in roughly 1 English sentence which rule and why it is different. Then give *two* example expressions: one where the formal semantics and your interpreter give answers and one where only your interpreter does.

(c) One of the rules can produce *more results* — it is nondeterministic where your interpreter is deterministic. Explain in roughly 1 English sentence which rule and why it is different. Then give a program where, thanks to this nondeterminism, the formal semantics can produce an answer but your interpreter would raise an exception.

Solution:
See next page

(a) Rule Replace: It always redistributes the pennies into nickels, whereas the interpreter may distribute the pennies into quarters, nickels, and dimes. Examples: They do the same for `ReplacePennies(MoneyConst(0,0,0,5)` but different for `ReplacePennies(MoneyConst(0,0,0,10)`.

(b) Rule Combine: The rule may only be applied when both sub-expressions return exactly the same value, a pre-condition the interpreter does not have. Provided the (spurious) pre-condition is met, Combine produces results consistent with the interpreter. Examples: They do the same for `CombineMoney(MoneyConst(1,2,3,4),MoneyConst(1,2,3,4))` but only the interpreter succeeds for `CombineMoney(MoneyConst(1,2,3,4),MoneyConst(1,2,3,5))`.

(c) Rule Half: The values of q', d', n', p' may be chosen non-deterministically provided they meet the constraint specified in the pre-condition whereas the interpreter implementation uses a deterministic algorithm. Example: `RemoveCoin (Half (0,0,0,50), Quarter)`, as we may choose $q' = 1$ (so the `RemoveCoin` term can be applied) whereas the interpreter will always choose $p' = 25$ so the `RemoveCoin` term fails. Another correct approach is where there are multiple ways to divide the money in half if using a non-subset of the given coins but no way from the given coins. For example `HalfValue(MoneyConst(1,0,0,1))` fails in the interpreter but can produce `(0,1,0,3)` or `(0,0,2,3)` as well as other possible results.

5. (**23** points) (Type Checking)

`exam.ml` defines part of an unusual and fairly misguided type system for the language we implemented in Problem 2. In this type system, each expression and variable is given a type of the form `even * even * even * even` where `type even = IsEven | MightNotBeEven`. Note that `IsEven` describes (only) numbers that are definitely even numbers, which includes 0.

We suppose the purpose of this type system is to prevent well-typed programs from causing `InterpFailure` exceptions from occurring when interpreted by `interp_large_coin_exp`, though the type system does a poor job of this.

(a) Complete the definition of `typecheck` which has type `(string * coin_type) list -> coin_exp -> coin_type` and which raises the exception `DoesNotTypecheck` for expressions that should not type-check. Three cases are given to you; do not change them. For the remaining cases:

- Do type-check subexpressions (of course) and/but do *not* use any more information about a subexpression other than its type. For example, notice that with the provided code `typecheck [] (CombineMoney (MoneyConst (0,1,0,1),MoneyConst (0,1,1,1)))` `= (IsEven,MightNotBeEven,MightNotBeEven,MightNotBeEven)` even though it could be determined "at compile time" that the number of dimes and pennies in the result is 2, which is even.
- There is no `IsOdd`, so even if you "know" a value must be odd, you have no choice but to use `MightNotBeEven`.
- `HalfValue` should require "knowing" that evaluating its subexpression will produce a result where all components are even numbers.
- Other than the previous points above, give the "best types" (most uses of `IsEven` and fewest uses of `DoesNotTypecheck`) you can.

(b) Give an example "program" of type `coin_exp` that demonstrates our "type system" is *unsound* given its stated purpose above. Do not use `Var` (so your example will work for any environment and heap).

(c) Give an example "program" of type `coin_exp` that demonstrates our "type system" is *incomplete* given its stated purpose above. Do not use `Var` (so your example will work for any environment and heap).

Solution:

i.
```
let rec typecheck env exp =
  let ifeven i = if i mod 2 = 0 then IsEven else MightNotBeEven in
  let merge_evens e1 e2 = match (e1,e2) with
                            (IsEven,IsEven) -> IsEven
                          | _ -> MightNotBeEven in
  let nothing_known =
    (MightNotBeEven, MightNotBeEven, MightNotBeEven, MightNotBeEven) in
  match exp with
    MoneyConst (q,d,n,p) -> (ifeven q, ifeven d, ifeven n, ifeven p)
  | Var s -> (try List.assq s env with Not_found -> raise DoesNotTypecheck)
  | CombineMoney (e1,e2) ->
    let (q1,d1,n1,p1) = typecheck env e1 in
    let (q2,d2,n2,p2) = typecheck env e2 in
    (merge_evens q1 q2,
     merge_evens d1 d2,
     merge_evens n1 n2,
```

```
     merge_evens p1 p2)
  | RemoveCoin (e,c) ->
    let (q,d,n,p) = typecheck env e in
    (match c with
      Quarter -> (MightNotBeEven,d,n,p)
    | Dime    -> (q,MightNotBeEven,n,p)
    | Nickel  -> (q,d,MightNotBeEven,p)
    | Penny   -> (q,d,n,MightNotBeEven))
  | HalfValue e ->
    let (q,d,n,p) = typecheck env e in
    if q = IsEven && d = IsEven && n = IsEven && p = IsEven
    then nothing_known
    else raise DoesNotTypecheck
  | ReplacePennies e ->
    ignore(typecheck env e);
    (MightNotBeEven, MightNotBeEven, MightNotBeEven, IsEven)
```

ii. `RemoveCoin ((MoneyConst (1,1,1,0), Penny)`

iii. `HalfValue (ReplacePennies (MoneyConst (0,0,0,50)))`

6. (**11** points) (Subtyping and References)

This problem considers adding mutable references (like OCaml's references) to our coin-expression language as well as subtyping on top of the type system from the previous problem. This problem should be done "in a text file" or similar (like Problem 4) since not all our additions will be "actually implemented in OCaml").

We make these additions:

- let-expressions of the form `let x : t = e1 in e2`, which are like in OCaml except we have an explicit type `t` on the variable *and* we allow `e1` to be a subtype of `t`.
- sequence-expressions `e1; e2` (as in OCaml)
- Expressions for creating and using references as in OCaml:
  - `ref e` to create a new reference initially containing the result of evaluating `e`
  - `!e` to evaluate `e` to a reference and produce its current contents
  - `e1 := e2` to evaluate `e1` to a reference and change its contents to the result of evaluating `e2`.
- Our type system now gives expressions and variables types that are defined by `coin_type'` where:
  `type coin_type' = MoneyType of coin_type | RefType of coin_type' | UnitType`
  and `coin_type` was defined in Problem 5.
- Like in OCaml, for any type (i.e., `coin_type'`) `t`, the reference operations have these types:
  - `ref e` has type `RefType t` if `e` has type `t`.
  - `!e` has type `t` if `e` has type `RefType t`.
  - `e1 := e2` has type `UnitType` if `e1` has type `RefType t` and `e2` has type `t`.

We assume this (*broken!*) definition of subtyping:

```
let rec subtype_proposed t1 t2 =
  let even_sub et1 et2 = (et1 = IsEven || et2 = MightNotBeEven) in
  match (t1,t2) with
    (MoneyType(qt1,dt1,nt1,pt1), MoneyType(qt2,dt2,nt2,pt2)) ->
      List.for_all2 even_sub [qt1;dt1;nt1;pt1] [qt2;dt2;nt2;pt2]
  | (RefType t1', RefType t2') -> subtype_proposed t1' t2'
  | (UnitType, UnitType) -> true
  | _ -> false
```

With all that set-up, here (finally!) are the questions:

(a) Fill in the blanks below so that this program type-checks and causes an `InterpFailure` exception when evaluated and relies on a "new" unsoundness caused by subtyping, not any unsoundness that was already present. In other words, provide two types (the first two blanks) and two expressions (the next two blanks) such that the program overall demonstrates a new cause of unsoundness.

```
let x : ______________ = ref (2,2,2,2) in
let y : ______________ = _____________ in
(_______________ ; HalfValue (!x))
```

(b) Explain in 1–3 English sentences how to change `subtype_proposed` to cause your answer to part (a) and all analogous examples not to type-check. Be specific about how you would change the `subtype_proposed` definition to fix it.

Solution:

(a)
```
let x : RefType(MoneyType(IsEven,IsEven,IsEven,IsEven)) = ref (2,2,2,2) in
let y : RefType(MoneyType(MightNotBeEven,IsEven,IsEven,IsEven)) = x in
( y := (1,2,2,2) ; HalfValue (!x))
```

(b) We need references to be invariant on types. The rule for deciding subtyping on `RefType` should compare the types `t1'` and `t2'` to require equality instead of recursively calling `subtype_proposed`. In particular, it can be: `subtype_proposed t1' t2' && subtype_proposed t2' t1'`.

7. (**6** points) (Haskell Warmup) In `exam.hs`, port from OCaml to Haskell the implementations of `all_coins_tree` and `penniless` from Problem 3 such that they have types `(Coin -> Bool) -> MoneyTree -> Bool` and `MoneyTree -> Bool` respectively.

   Note we are *not* asking you to port `all_coins_tree_cps` nor `penniless2` (though it's not difficult).

   Solution:

```
all_coins_tree :: (Coin -> Bool) -> MoneyTree -> Bool
all_coins_tree f t =
  case t of
    Leaf c -> f c
    Node c t1 t2 -> f c && all_coins_tree f t1 && all_coins_tree f t2

penniless :: MoneyTree -> Bool
penniless = all_coins_tree (\ c -> case c of
                                     Penny -> False
                                     _ -> True)
```

8. (**10** points) (Haskell IO)

Continue working in `exam.hs`:

(a) Implement `n_times` to take an IO action a and a number n and produce an IO action that, when performed, performs a a total of n times (and ignores the results). Your function should have type `IO a -> Int -> IO ()` *or a more general type*. Assume $n \geq 0$.

(b) Use `n_times` and the standard library's `putStr` to implement `printMoney :: Money -> IO ()`, which should, given the value `Money q d n p`, produce an IO action that, when performed, behaves as follows:

- If `q`, `d`, `n`, and `p` are all 0, then it prints *you're broke!*.
- Otherwise it prints *quarter* followed by a space `q` times then prints *dime* followed by a space `d` times then prints *nickel* followed by a space `n` times then prints *penny* followed by a space `p` times. (Yes, this prints a trailing space at the end; that's fine for an exam.)

Solution:

```
n_times :: IO a -> Int -> IO ()
n_times a n = if n == 0 then return () else a >> n_times a (n-1)

printMoney :: Money -> IO ()
printMoney (Money 0 0 0 0) = putStr "you're broke!"
printMoney (Money q d n p) =
  do {
    n_times (putStr "quarter ") q ;
    n_times (putStr "dime ")    d ;
    n_times (putStr "nickel ")  n ;
    n_times (putStr "penny ")   p
    }
```

9. (**10** points) (More Haskell)

The code in `exam.hs` includes this instance declaration:

```
instance Eq Money where
  (==) (Money q1 d1 n1 p1) (Money q2 d2 n2 p2) =
    q1 == q2 && d1 == d2 && n1 == n2 && p1 == p2
```

as well as some sample tests that use this definition.

(a) In `exam.hs`, *comment out* the definition of `(==)` and provide a different definition such that money values are equal if they have the "same money value."

(b) (In either a separate text file or as comment in the Haskell file), explain in roughly 3–4 English sentences *how* `main` behaves *both* before and after this change and *why* it behaves how it does.

(c) (In either a separate text file or as comment in the Haskell file), explain in 1–2 English sentences how your answer in part (b) would differ in a (hypothetical) variant of OCaml with typeclasses.

Solution:

(a)
```
  instance Eq Money where
  (==) (Money q1 d1 n1 p1) (Money q2 d2 n2 p2) =
--    q1 == q2 && d1 == d2 && n1 == n2 && p1 == p2
    25 * q1 + 10 * d1 + 5 * n1 + p1 == 25 * q2 + 10 * d2 + 5 * n2 + p2
```

(b) The third comparison, `print (reduce_pennies_to_half (Money 2 3 4 15) == reduce_pennies_to_half (Money 4 0 1 0))` does not throw a divide by zero exception under the old definition. The evaluation of the penny component in `reduce_pennies_to_half` that may throw the divide by zero error is not executed immediately: it is suspended until needed. This suspended thunk is not forced under the old definition of equality; the `==` operator finds the two quarter terms unequal and returns immediately (thanks to the short-circuiting nature of the `&&` operator). However, the updated definition of equality requires forcing all components of the compared Money terms leading to a divide by zero.

(c) OCaml is a strict language and would therefore eagerly evaluate all components of the Money constructor in `reduce_pennies_to_half`. Thus, both implementations will throw a divide by zero error.

10. (**12** points) (Object-Oriented Programming)

Consider the skeleton below of a class definition using the same sort of pseudocode from lecture. This class for "money" has methods that correspond to analogous functions we wrote in OCaml or Haskell, where, like in the rest of the exam, we avoid mutation — in this case by having `removeCoin` return a new object instead:

```
class Money {
  private int num_quarters, num_dimes, num_nickels, num_pennies; // 4 private fields

  constructor(int q, int d, int n, int p) { ... }

  int getQuarters() { num_quarters }
  // similar "getters" for dimes, nickels, and pennies [not shown]

  int valueOfMoney() { 25 * num_quarters + 10 * num_dimes + ... }

  // return a new object that is almost like "self" with one less coin
  Money removeCoin(Coin c) { ... }

  // means same number of each kind of coin; NOT same valueOfMoney
  bool equals(Money other) { ... }
}
```

A common argument in favor of OOP is that subclassing and subtyping make software more reusable and extensible. Suppose in this case we wish to create a subclass that supports dollar-coins:

```
// assume "MoreCoins" has the usual coins *and* dollar coins
// (you can assume MoreCoins is a subclass of Coin or just a different
// type -- either assumption doesn't really change the questions below)
class MoreMoney extends Money { // subclass supporting dollar coins
   private int num_dollars; // add a field for dollar coins

   int getDollars() { num_dollars } // new getter

   constructor(List<MoreCoins> coinlist) {  ... }
   ...
}
```

In a text file or similar, for each of the following, either describe a problem with it in approximately 1-2 precise English sentences (in terms of functionality and/or type-checking) or if there are no problems, just say "works fine" (without any explanation needed).

(a) `MoreMoney` inheriting `getQuarters` from `Money`

(b) `MoreMoney` inheriting `valueOfMoney` from `Money`

(c) `MoreMoney` inheriting `removeCoin` from `Money`

(d) `MoreMoney` inheriting `equals` from `Money`

(e) `MoreMoney` overriding `getQuarters` from `Money`

(f) `MoreMoney` overriding `valueOfMoney` from `Money`

(g) `MoreMoney` overriding `removeCoin` from `Money`

(h) `MoreMoney` overriding `equals` from `Money`

Solution:

(a) Works fine (want same behavior in subclass)

(b) Type-checks, but it ignores dollar coins which is not what we want.

(c) Does not work well since the Coin superclass does not “know” about the `num_dollar` field, so the returned value will drop any dollar coins.

(d) Problematic: the implementation will ignore dollar coins.

(e) Works fine but there’s no point: due to data hiding the implementation would have to dispatch to the super implementation.

(f) Works fine — we can add the `100*num_dollars` term to the value returned from the superclass implementation.

(g) Works fine as we can return a new MoreMoney. However, the argument type still has to be `Coin` instead of `MoreCoin` due to contravariance on method arguments. In practice, the subclass implementation would likely contain an `instanceof` check.

(h) Does not work well as we can’t force the argument `other` to be MoreMoney (contravariance). Further, considering the class of `other` in the overridden equals method will break symmetry or transitivity.