Computer Science & Engineering 505
Concepts of Programming Languages
Autumn 1994
Sample Solution — Assignment 7

1.1 The following program runs successfully but fails the typchecker:

```
if true then 7 else false
```

When it runs, it just returns 7. The problem is that there is no type τ such that 7 and false are both of type τ. (See Cardelli, bottom of page 11.)

1.2 Yes, the semantic function can be modified so that a program will pass the typechecker in exactly those cases where it does not output a value of "wrong," without altering the type inference rules. Intuitively, all we need to do is introduce a runtime type check that does the same thing as the typechecker does.

In more detail, the specific problem is in the part of the function that describes conditionals (see question 1.1). Informally, the modified rule says:

> The meaning of a conditional expression is *wrong* if the test isn't a boolean, or if the join of the types of the *then* and *else* parts is undefined, i.e. if there is no type τ such that the *then* and *else* parts are both of type τ. Otherwise, if the value of the test is true, the result is the meaning of the *then* part; if the value of the test is false, the result is the meaning of the *else* part.

Formally:
$\mathcal{E}\,[\![\mathit{if}\ e\ \mathit{then}\ e'\ \mathit{else}\ \ e'']\!]\,v =$
$\mathit{if}\ \mathcal{E}\,[\![e]\!]\,v \in T\ \mathit{and}\ \mathcal{T}\,[\![e']\!]\,\mu \uparrow \mathcal{T}\,[\![e'']\!]\,\mu\ \mathit{is\ defined\ for\ some}\ \mu\ \mathit{which\ agrees\ with\ some}\ A\ \mathit{which\ agrees\ with}\ v$
then
$(\mathit{if}\ (\mathcal{E}\,[\![e]\!]\,v \mid T)\ \ \mathit{then}\ \mathcal{E}\,[\![e']\!]\,v\ \mathit{else}\ \mathcal{E}\,[\![e'']\!]\,v)$
$\mathit{else\ wrong}$

1.3 No, there isn't a way to modify the type function on page 11 and the inference rules given on page 12 so that a program will pass the typechecker in exactly those cases where it does not output a value of "wrong". Consider the program

```
if boolean_expr then 7 else false+5
```

If `boolean_expr` has the value true, then the program should pass the typechecker; if it has the value false, then the program shouldn't. But we can't decide at compile time whether an arbitrary expression has value true or false.

2. The place where call-by-value semantics is specified is in the part of the semantic function defining function application. To give call-by-name semantics, just delete the test to see whether the argument is bottom or wrong. So the new equation is:
$\mathcal{E}\,[\![e\,e']\!]\,v = \mathit{if}\ \mathcal{E}\,[\![e]\!]\,v \in F\ \mathit{then}\ (\mathcal{E}\,[\![e]\!]\,v \mid F)\,(\mathcal{E}\,[\![e']\!]\,v)\ \ \mathit{else\ wrong}$

3. Under the contravariant rule, a read-only array of strings is a subtype of a read-only array of Object. (By "read-only array" we mean an array with an `at:` method, but not an `at:put:` method. However, an writable array of strings is *not* a subtype of writable array of Object. (See the lecture notes on contravariance.) So our type system should be flexible enough to let us specify that the type expected is read-only array of Object, rather than just array of Object. (In this case, the writable array of strings is a subtype of read-only array of Object.)

4.

```
generic protocol BinaryTree2[T]
  add: <T>
  remove: <T> ifAbsent: <Block[^X]> ^<T|X> where X:: actual arg: 2 returnType
  includes: <T>
  isEmpty ^<Boolean>
  max ^<T>
  printOn: <WriteStream[Character]>
  do: <Block[T]>
  inorderDo: <Block[BinaryNode[T]]>
  postorderDo: <Block[BinaryNode[T]]>
  preorderDo: <Block[BinaryNode[T]]>

generic protocol BinaryNode[T]
  max ^<T>
  add: <T>
  remove: <T> ifAbsent: <Block[^X]> ^<T|X> where X:: actual arg: 2 returnType
  setValue: <T>
  includes: <T>
  value ^<T>
  inorderDo: <Block[BinaryNode[T]]>
  postorderDo: <Block[BinaryNode[T]]>
  preorderDo: <Block[BinaryNode[T]]>
```

5. Structural equivalence for types in Algol-68 was done based on the internal representation of the types – there was no information hiding or data abstraction. For example, the following two record types would be considered as equivalent:

```
type SackOfFlour is record pounds: real; end record;
type EnglishBankAccount is record pounds: real; end record;
```

In contrast, type equivalence in Strongtalk is based on the external message interface that the object provides, so that structural equivalence is more reasonable. In addition, one can obtain the benefits of name equivalence by adding a *brand* (a tag) to a protocol in Strongtalk. For example, if we had two classes, `SackOfFlour` and `EnglishBankAccount` that both respond to the `pounds` message, we could still add a brand to them to keep the protocols distinct.