The University of Nottingham

SCHOOL OF COMPUTER SCIENCE

A LEVEL 4 MODULE, SPRING SEMESTER 2011–2012

MATHEMATICAL FOUNDATIONS OF PROGRAMMING

ANSWERS

Time allowed TWO hours

Candidates may complete the front cover of their answer book and sign their desk card but must NOT write anything else until the start of the examination period is announced.

Answer QUESTION ONE and THREE other questions

No calculators are permitted in this examination.

Dictionaries are not allowed with one exception. Those whose first language is not English may use a standard translation dictionary to translate between that language and English provided that neither language is the subject of this examination. Subject-specific translation directories are not permitted.

No electronic devices capable of storing and retrieving text, including electronic dictionaries, may be used.

Note: ANSWERS

Question 1 (Compulsory)

(a) See appendix A for the grammars and operational semantics relevant to this question. (You do not need the typing rules for this question.) Use the operational semantics to evaluate the following term until a value is obtained:

$$\begin{array}{l}\texttt{if (if (iszero (succ 0))}\\ \qquad\quad\texttt{then (iszero (succ 0))}\\ \qquad\quad\texttt{else (iszero (pred 0)))}\\ \quad\ \texttt{then (succ 0)}\\ \quad\ \texttt{else 0}\end{array}$$

The validity of *the first two* evaluation steps should be proved by applying the rules of the operational semantics. For the remaining steps, just show the sequence of terms in the right order. (8)

Answer: *Straightforward, will only sketch the answer. The sequence of evaluation steps is as follows:*

$$\begin{array}{ll}\multicolumn{2}{l}{\texttt{if (if (iszero (succ 0)) then (iszero (succ 0)) else (iszero (pred 0)))}}\\ \multicolumn{2}{l}{\quad\texttt{then (succ 0)}}\\ \multicolumn{2}{l}{\quad\texttt{else 0}}\\ \longrightarrow & \texttt{if (if false then (iszero (succ 0)) else (iszero (pred 0)))}\\ & \quad\texttt{then (succ 0)}\\ & \quad\texttt{else 0}\\ \longrightarrow & \texttt{if (iszero (pred 0)) then (succ 0) else 0}\\ \longrightarrow & \texttt{if (iszero 0) then (succ 0) else 0}\\ \longrightarrow & \texttt{if true then (succ 0) else 0}\\ \longrightarrow & \texttt{succ 0}\end{array}$$

For full marks, the first two steps should be proved using the rules of the operational semantics. As an example, the first step should be proved as follows:

$$\dfrac{\dfrac{\dfrac{}{\texttt{iszero (succ 0)} \longrightarrow \texttt{false}}\ \text{E-ISZEROSUCC}}{\begin{array}{l}\texttt{if (iszero (succ 0)) then (iszero (succ 0)) else (iszero (pred 0))}\\ \longrightarrow \texttt{if false then (iszero (succ 0)) else (iszero (pred 0))}\end{array}}\ \text{E-IF}}{\begin{array}{l}\texttt{if (if (iszero (succ 0)) then (iszero (succ 0)) else (iszero (pred 0)))}\\ \quad\texttt{then (succ 0)}\\ \quad\texttt{else 0}\\ \longrightarrow \texttt{if (if false then (iszero (succ 0)) else (iszero (pred 0)))}\\ \qquad\texttt{then (succ 0)}\\ \qquad\texttt{else 0}\end{array}}\ \text{E-IF}$$

(b) Identify the β-redexes and the free variable occurrences in the following λ-calculus term:

$$\lambda x.((\lambda y.(x\ (\lambda y.u\ v))\ y)\ (((\lambda z.z)\ (\lambda z.y))\ x))$$

(5)

Answer: *Each β-redex is underlined and each free variable occurrence enclosed in a box:*

$$\lambda x.\underline{((\lambda y.(x\ (\lambda y.\boxed{u}\ \boxed{v}))\ y)\ (\underline{((\lambda z.z)\ (\lambda z.\boxed{y}))}\ x))}$$

(There is also an η-redex.)

(c) Given the definitions:

$$\begin{aligned} \mathrm{I} &\equiv \lambda x.x \\ \mathrm{K} &\equiv \lambda x.\lambda y.x \\ \omega &\equiv \lambda x.x\ x \end{aligned}$$

reduce the following λ-calculus term to normal form if possible:

$$(\mathrm{I}\ \mathrm{K})\ (\mathrm{I}\ \mathrm{I})\ (\omega\ \omega)$$

Show each unfolding step (expansion of a definition) and each individual β-reduction step. If it is not possible to reach a normal form, explain why. (5)

Answer: *Normal-order reduction ensures a normal form will be found, if it exists. So carry out normal-order reduction (outermost, left-most redex first) until either a normal form has been reached, or it becomes evident that reduction isn't going to yield a normal form:*

$$\begin{aligned} &(\mathrm{I}\ \mathrm{K})\ (\mathrm{I}\ \mathrm{I})\ (\omega\ \omega) \\ &\quad \equiv\ ((\lambda x.x)\ \mathrm{K})\ (\mathrm{I}\ \mathrm{I})\ (\omega\ \omega) \\ &\quad \underset{\beta}{\rightarrow}\ \mathrm{K}\ (\mathrm{I}\ \mathrm{I})\ (\omega\ \omega) \\ &\quad \equiv\ (\lambda x.\lambda y.x)\ (\mathrm{I}\ \mathrm{I})\ (\omega\ \omega) \\ &\quad \underset{\beta}{\rightarrow}\ (\lambda y.(\mathrm{I}\ \mathrm{I}))\ (\omega\ \omega) \\ &\quad \underset{\beta}{\rightarrow}\ \mathrm{I}\ \mathrm{I} \\ &\quad \equiv\ (\lambda x.x)\ \mathrm{I} \\ &\quad \underset{\beta}{\rightarrow}\ \mathrm{I} \\ &\quad \equiv\ \lambda x.x \end{aligned}$$

As there are no more β-redexes, a normal form has been reached and we are done.

(d) Explain what is meant by "well-typed programs do not go wrong". Your answer should define any key technical terms used, clarify the relation to the notions of *progress* and *preservation*, and should specifically discuss the possibility of run-time errors and non-termination. (7)

Answer: *"Well-typed programs do not go wrong" means that it is guaranteed that a well-typed program will not get* stuck*; i.e., end up in an ill-defined semantic configuration. This breaks down into two parts:* progress*, which means that a well-typed program either has been evaluated to a result or that it can be evaluated further, and* preservation*, which means that well-typedness is invariant under evaluation. However, it does not in general rule out run-time errors such as division by zero or out-of-bound array indices. But it does ensure that the semantics explicitly accounts for such possibilities. Also, there are, in general, no termination guarantees: the evaluation of a well-typed program may loop.*

Question 2
See appendix A for the grammar and operational semantics relevant to this question. (You do not need the typing rules for this question.)

Add an exception mechanism to the language. The syntax for the new constructs are given below. You should add all necessary rules to the operational semantics in order to formally define the meaning of the new constructs.

New language constructs:

$$\begin{array}{lll@{\qquad}r}
t & \rightarrow & \ldots & \textit{terms:} \\
 & | & \texttt{excn}\ nv & \textit{exception} \\
 & | & \texttt{raise}\ t & \textit{raising an exception} \\
 & | & \texttt{try}\ t\ \texttt{catch any with}\ t & \textit{catching an arbitrary exception} \\
 & | & \texttt{try}\ t\ \texttt{catch}\ t\ \texttt{with}\ t & \textit{catching only a specific exception}
\end{array}$$

The idea is that `excn` *nv* is a new normal form (but *not* a value!) standing for an exception. Note that an exception carries a numerical value so that different exceptions can be told apart.

The meaning of the construct `raise` t is that the argument t first is evaluated and (assuming the argument evaluates to a numerical value) the entire construct then evaluates to an exception carrying this value.

If an exception is the result of evaluation *anywhere* in a term, with the exception of the `try`-construct as described below, then that entire term should evaluate to the exception. This means that a number of *exception propagation rules* have to be added to the semantics for the original language constructs. They are as follows:

$$\texttt{if}\ (\texttt{excn}\ nv_1)\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \longrightarrow \texttt{excn}\ nv_1 \qquad \text{(E-IFEXCN)}$$

$$\texttt{succ}\ (\texttt{excn}\ nv_1) \longrightarrow \texttt{excn}\ nv_1 \qquad \text{(E-SUCCEXCN)}$$

$$\texttt{pred}\ (\texttt{excn}\ nv_1) \longrightarrow \texttt{excn}\ nv_1 \qquad \text{(E-PREDEXCN)}$$

$$\texttt{iszero}\ (\texttt{excn}\ nv_1) \longrightarrow \texttt{excn}\ nv_1 \qquad \text{(E-ISZEROEXCN)}$$

Note that propagation rules also will be needed for the new language constructs; for example, if the evaluation of the argument to `raise` happens to raise an exception itself.

Finally, the `try`-construct allows exceptions to be caught. There are two forms. The first one allows *any* exception to be caught. The meaning of `try` t_1 `catch any with` t_2 is that if t_1 evaluates to a value, then that is the result of the entire construct. However, if t_1 evaluates to an exception, then the overall result is given by t_2.

The second form `try` t_1 `catch` t_2 `with` t_3 is similar to the first one, except that an exception `excn` nv_1 is only caught if t_2 evaluates to the same numerical value nv_1. If t_2 evaluates to a different numerical value, the exception should be re-raised; i.e., the entire construct should evaluate to `excn` nv_1.

(a) Give semantic rules for `raise` t_1 that capture the behaviour described above. (5)

(b) Give semantic rules for `try` t_1 `catch any with` t_2 that capture the behaviour described above. (5)

(c) Give semantic rules for `try` t_1 `catch` t_2 `with` t_3 that capture the behaviour described above. You can assume that equality and inequality relations are defined for values; i.e., feel free to use premises like $nv_1 = nv_2$ or $nv_1 \neq nv_2$ in the rules if you need to. (8)

(d) Using your semantic rules, evaluate:

```
if (try (raise (succ (raise 0))) catch (succ 0) with true)
   then (succ 0)
   else 0
```

For full marks, the first step should be formally proved using the semantics rules. For the remaining steps, just show the sequence of terms. (7)

Answer:

(a) Semantic rules for `raise`*:*

$$\texttt{raise}\ nv_1 \longrightarrow \texttt{excn}\ nv_1 \qquad \text{(E-RAISENV)}$$

$$\texttt{raise}\ (\texttt{excn}\ nv_1) \longrightarrow \texttt{excn}\ nv_1 \qquad \text{(E-RAISEEXCN)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{raise}\ t_1 \longrightarrow \texttt{raise}\ t_1'} \qquad \text{(E-RAISE)}$$

(b) Semantic rules for `try` *catching any exception:*

$$\texttt{try}\ v_1\ \texttt{catch any with}\ t_2 \longrightarrow v_1 \qquad \text{(E-TRYANYV)}$$

$$\texttt{try}\ (\texttt{excn}\ nv_1)\ \texttt{catch any with}\ t_2 \longrightarrow t_2 \qquad \text{(E-TRYANYEXCN)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{try}\ t_1\ \texttt{catch any with}\ t_2 \longrightarrow \texttt{try}\ t_1'\ \texttt{catch any with}\ t_2} \qquad \text{(E-TRYANY)}$$

(c) Semantic rules for `try` *catching specific exception only:*

$$\texttt{try}\ v_1\ \texttt{catch}\ t_2\ \texttt{with}\ t_3 \longrightarrow v_1 \qquad \text{(E-TRYV)}$$

$$\texttt{try}\ (\texttt{excn}\ nv_1)\ \texttt{catch}\ nv_1\ \texttt{with}\ t_3 \longrightarrow t_3 \qquad \text{(E-TRYEXCNMATCH)}$$

$$\frac{nv_1 \neq nv_2}{\texttt{try}\ (\texttt{excn}\ nv_1)\ \texttt{catch}\ nv_2\ \texttt{with}\ t_3 \longrightarrow \texttt{excn}\ nv_1} \qquad \text{(E-TRYEXCNNOMATCH)}$$

$$\begin{array}{l}\texttt{try}\ (\texttt{excn}\ nv_1)\ \texttt{catch}\ (\texttt{excn}\ nv_2)\ \texttt{with}\ t_3\\ \longrightarrow \texttt{excn}\ nv_2\end{array} \qquad \text{(E-TRYEXCNEXCN)}$$

$$\frac{t_2 \longrightarrow t_2'}{\begin{array}{l}\texttt{try}\ (\texttt{excn}\ nv_1)\ \texttt{catch}\ t_2\ \texttt{with}\ t_3\\ \longrightarrow \texttt{try}\ (\texttt{excn}\ nv_1)\ \texttt{catch}\ t_2'\ \texttt{with}\ t_3\end{array}} \qquad \text{(E-TRYEXCN)}$$

$$\frac{t_1 \longrightarrow t_1'}{\begin{array}{l}\texttt{try}\ t_1\ \texttt{catch}\ t_2\ \texttt{with}\ t_3\\ \longrightarrow \texttt{try}\ t_1'\ \texttt{catch}\ t_2\ \texttt{with}\ t_3\end{array}} \qquad \text{(E-TRY)}$$

(d) The first evaluation step is justified as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{raise 0} \longrightarrow \texttt{excn 0}}\ \text{E-RAISENV}}{\texttt{succ (raise 0)} \longrightarrow \texttt{succ (excn 0)}}\ \text{E-SUCC}}{\texttt{raise (succ (raise 0))} \longrightarrow \texttt{raise (succ (excn 0))}}\ \text{E-RAISE}}{\begin{array}{l}\texttt{try (raise (succ (raise 0))) catch (succ 0) with true}\\ \longrightarrow \texttt{try (raise (succ (excn 0))) catch (succ 0) with true}\end{array}}\ \text{E-TRY}}{\begin{array}{l}\texttt{if (try (raise (succ (raise 0))) catch (succ 0) with true)}\\ \quad \texttt{then (succ 0)}\\ \quad \texttt{else 0}\\ \longrightarrow \texttt{if (try (raise (succ (excn 0))) catch (succ 0) with true)}\\ \qquad \texttt{then (succ 0)}\\ \qquad \texttt{else 0}\end{array}}\ \text{E-IF}$$

The complete sequence of evaluation steps is:

```
if (try (raise (succ (raise 0))) catch (succ 0) with true)
   then (succ 0)
   else 0
  ⟶  if (try (raise (succ (excn 0))) catch (succ 0) with true)
        then (succ 0)
        else 0
  ⟶  if (try (raise (excn 0)) catch (succ 0) with true)
        then (succ 0)
        else 0
  ⟶  if (try (excn 0) catch (succ 0) with true)
        then (succ 0)
        else 0
  ⟶  if (excn 0) then (succ 0) else 0
  ⟶  excn 0
```

Question 3
This question concerns the pure, untyped λ-calculus, enriched with λ-definable constants where indicated.

(a) Consider the following recursive definition of a function for computing the nth Fibonacci number ($n \in \mathbb{N}$ assumed):

```
fib(n) = if n == 0 then
             0
         else
             if n == 1 then
                 1
             else
                 fib(n-1) + fib(n-2)
```

Show how to translate this function into the pure λ-calculus, explaining the key ideas of the translation. You may assume normal-order evaluation and use the following λ-definable constants:

IF	Usual three-argument conditional; first argument, the condition, assumed to be of Boolean type
EQ	Comparison for numerical equality
PLUS	Numerical addition
MINUS	Numerical subtraction
$0, 1, 2, \ldots$	Numerical constants; any natural number you need

Any other constants needed in your translation have to be defined and explained. (10)

Answer: *The idea of the translation is to abstract out the recursively called function as an extra, first, argument. The resulting residual function is thus a function that will return a function that computes the nth Fibonacci number if applied to a function that computes the nth Fibonacci number. Or, in other words, the desired function is the* fixed point *of the residual function. If we introduce a fixed-point combinator* Y *for computing fixed points, the above function* ***fib*** *can be translated into the λ-calculus as follows, where* FIB′ *is the residual and* FIB *the desired translation:*

$$\begin{aligned}
\mathrm{FIB}' &\equiv \lambda f.\lambda n.\mathrm{IF}\ (\mathrm{EQ}\ n\ 0) \\
&\qquad\qquad 0 \\
&\qquad\qquad (\mathrm{IF}\ (\mathrm{EQ}\ n\ 1) \\
&\qquad\qquad\quad 1 \\
&\qquad\qquad\quad (\mathrm{PLUS}\ (f\ (\mathrm{MINUS}\ n\ 1))\ (f\ (\mathrm{MINUS}\ n\ 2)))) \\
\mathrm{FIB} &\equiv \mathrm{Y}\ \mathrm{FIB}'
\end{aligned}$$

Y *is the call-by-name fixed point combinator and works under call-by-name or normal-order evaluation. It has the following definition:*

$$\mathrm{Y} \quad \equiv \quad \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

What makes this work is the fact that Y *satisfies the fixed point equation*

$$\mathrm{Y\ F} = \mathrm{F\ (Y\ F)}$$

In other words, Y *ensures that a function* F *to which* Y *is applied gets applied to the fixed point of the function* F *itself, which is computed by applying* Y *to* F*.*

Another way to understand this is that Y *enables a recursive definition to be unfolded on demand, thus simulating the jump to a specific code sequence, which is how function calls and recursion usually are handled, by inlining a copy of the called function at the call site.*

(b) Show how triples (ternary products) can be encoded in the pure λ-calculus without assuming any existing definitions. That is, define a constructor function for building a triple, and three projection functions for selecting the first, second, and third component of a triple, respectively. (4)

Answer:

$$\begin{aligned}
\mathrm{TRIPLE} &\equiv \lambda f.\lambda s.\lambda t.\lambda b.b\ f\ s\ t \\
\mathrm{FST} &\equiv \lambda p.p\ (\lambda f.\lambda s.\lambda t.f) \\
\mathrm{SND} &\equiv \lambda p.p\ (\lambda f.\lambda s.\lambda t.s) \\
\mathrm{THD} &\equiv \lambda p.p\ (\lambda f.\lambda s.\lambda t.t)
\end{aligned}$$

(c) Explain the problem of *name capture*. Illustrate your answer with an example. (4)

Answer: *Name capture occurs when a term with free variables is naively substituted for a variable in a context where one or more of the variables that are free in the term are bound. For example, if the application*

$$(\lambda x.\lambda y.x)\ y$$

were to be reduced naively by substituting y for x in $\lambda y.x$, then the result would be $\lambda y.y$ which is wrong. The initially free y has been confused with a different bound variable that just happened to have the same name.

(d) State *Church-Rosser theorems I and II*, and put them into context by briefly discussing their implications and the pros and cons of *normal-order* vs. *call-by-value* evaluation. Also briefly explain the idea of *lazy* evaluation against this background. (7)

Answer:

- ***Church-Rosser Theorem I:*** *For all λ-calculus terms t, t_1, and t_2 such that $t \underset{\beta}{\overset{*}{\rightarrow}} t_1$ and $t \underset{\beta}{\overset{*}{\rightarrow}} t_2$, there exists a term t_3 such that $t_1 \underset{\beta}{\overset{*}{\rightarrow}} t_3$ and $t_2 \underset{\beta}{\overset{*}{\rightarrow}} t_3$. (That is, β-reduction is confluent.)*
- ***Church-Rosser Theorem II:*** *For all λ-calculus terms t_1 and t_2, if $t_1 \underset{\beta}{\overset{*}{\rightarrow}} t_2$ and t_2 is a normal form (no redexes), then t_1 reduces to t_2 under normal-order reduction.*

This means that normal-order reduction has the best possible termination properties; i.e., if a normal form exists, it will be found through normal order reduction, whereas other reduction strategies may diverge ("loop" for ever). Moreover, due to confluence, the normal form is unique. However, if a term is successfully reduced to a normal form using a different reduction strategy, this may well be accomplished in fever reduction steps than when using normal order. In fact, this is quite common as redexes often are duplicated under normal order reduction, which can lead to duplication of work. Lazy evaluation is an optimised implementation with normal order reduction semantics *where duplication is avoided by sharing of redexes: if reduced, the redex is overwritten by the result. Thus, under lazy evaluation, any one redex is reduced at most once.*

Question 4

See appendix A for the grammar, operational semantics, and typing rules relevant to this question.

(a) The expression language defined in appendix A is to be extended with lists of elements of some specific type (i.e., homogeneous lists) as follows:

$$\begin{array}{lll r}
t & \rightarrow & & \textit{terms:} \\
 & \ldots & \ldots & \ldots \\
 & \mid & \texttt{null} & \textit{empty list} \\
 & \mid & \texttt{cons}\ t\ t & \textit{list construction} \\
 & & & \\
v & \rightarrow & & \textit{values:} \\
 & \ldots & \ldots & \ldots \\
 & \mid & \texttt{null} & \textit{null value} \\
 & \mid & \texttt{cons}\ v\ v & \textit{list value} \\
 & & & \\
T & \rightarrow & & \textit{types:} \\
 & \ldots & \ldots & \ldots \\
 & \mid & \texttt{List}\ T & \textit{list of elements of type } T
\end{array}$$

Provide (call-by-value) evaluation rules (where needed) and typing rules for each of the new term constructs in the same style as the existing rules. The constant `null` has type `List` T for some type T, `cons` takes a value of some type T and a list of elements of the same type T, i.e. `List` T, as arguments and returns a list of type `List` T. (5)

Answer:

$$\frac{t_1 \longrightarrow t_1'}{\texttt{cons}\ t_1\ t_2 \longrightarrow \texttt{cons}\ t_1'\ t_2} \quad \text{(E-CONS1)}$$

$$\frac{t_2 \longrightarrow t_2'}{\texttt{cons}\ v_1\ t_2 \longrightarrow \texttt{cons}\ v_1\ t_2'} \quad \text{(E-CONS2)}$$

$$\texttt{null} : \texttt{List}\ T \quad \text{(T-NULL)}$$

$$\frac{t_1 : T \quad t_2 : \texttt{List}\ T}{\texttt{cons}\ t_1\ t_2 : \texttt{List}\ T} \quad \text{(T-CONS)}$$

(Evaluation of the arguments to `cons` in the other order is also fine.)

(b) *Progress* can be formulated as follows:

> THEOREM [PROGRESS]: Suppose that t is a well-typed term (i.e., $t : T$), then either t is a value or else there is some t' with $t \longrightarrow t'$.

Prove Progress for the cases where the last step of a derivation was by the typing rule for `null` (call it T-NULL) and when it was by the rule for `cons` (call it T-CONS) by induction on the structure of a typing derivation. (10)

Answer:

Case T-NULL: $t = \texttt{null}$ *But then* t *is a value, so the theorem holds trivially.*

Case T-CONS: $t = \texttt{cons}\, t_1\, t_2$
$t_1 : T \quad t_2 : \texttt{List}\, T$

By the induction hypothesis, either t_1 *and* t_2 *are values, or else there is some* t_1' *such that* $t_1 \longrightarrow t_1'$ *and/or some* t_2' *such that* $t_2 \longrightarrow t_2'$.

If both t_1 *and* t_2 *are values, then so is* t*, and the theorem holds.*

On the other hand, if $t_1 \longrightarrow t_1'$*, then by E-CONS1,* $t \longrightarrow \texttt{cons}\ t_1'\ t_2$*, and the theorem holds.*

Finally, if $t_1 = v_1$ *is a value, but* t_2 *not, then it must be the case that* $t_2 \longrightarrow t_2'$*. Then, by E-CONS2, we have* $t \longrightarrow \texttt{cons}\ v_1\ t_2'$*, and the theorem holds.*

(c) *Preservation* can be formulated as follows:

> THEOREM [PRESERVATION]: If $t : T$ and $t \longrightarrow t'$ then $t' : T$.

Prove Preservation for the cases T-NULL and T-CONS by induction on the structure of a typing derivation. (10)

Answer: *Case T-NULL:* $t = \texttt{null}$ *There is no evaluation rule for* `null`*, by design as* `null` *is a value, so the theorem holds trivially.*

Case T-CONS: $t = \texttt{cons}\, t_1\, t_2$
$t_1 : T \quad t_2 : \texttt{List}\, T$
$t \longrightarrow t'$ *for some* t'

The only possible evaluation rules are one of E-CONS1 and E-CONS2.

If evaluation is by E-CONS1, then we know $t1 \longrightarrow t1'$*. As we also know* $t_1 : T$*, we can apply the induction hypothesis and conclude that* $t_1' : T$*. And then, by T-CONS, we can conclude* $\texttt{cons}\, t1'\ t2 : \texttt{List}\, T$.

If evaluation is by E-CONS2, then we know $t2 \longrightarrow t2'$*. As we also know* $t_2 : \texttt{List}\, T$*, we can apply the induction hypothesis and conclude that* $t_2' : \texttt{List}\, T$*. And then, by T-CONS, we can conclude* $\texttt{cons}\, t1\ t2' : \texttt{List}\, T$.

Thus, in both cases, the resulting term is well-typed and the evaluation has moreover preserved the type, proving that the theorem holds for the T-CONS case.

Question 5

(a) The following is a variant of the Polymorphic λ-calculus or System F with `Nat`, the natural numbers, as a base type:

$$
\begin{array}{lcll}
T & \to & & \textit{types:} \\
 & & \texttt{Nat} & \textit{natural number type} \\
 & | & T \to T & \textit{function type} \\
 & | & X & \textit{type variable} \\
 & | & \forall X \,.\, T & \textit{universally quantified type} \\
\\
\Gamma & \to & & \textit{contexts:} \\
 & & \emptyset & \textit{empty context} \\
 & | & \Gamma,\, x : T & \textit{context extended with variable typing} \\
 & | & \Gamma,\, X & \textit{context extended with type variable} \\
\\
t & \to & & \textit{terms:} \\
 & & n & \textit{natural number, } n \in \mathbb{N} \\
 & | & x & \textit{variable} \\
 & | & \lambda x{:}T \,.\, t & \textit{abstraction} \\
 & | & t\ t & \textit{application} \\
 & | & \Lambda X \,.\, t & \textit{type abstraction} \\
 & | & t\ [T] & \textit{type application}
\end{array}
$$

The ternary relation $\Gamma \vdash t : T$ says that expression t has type T in the type context Γ and it is defined by the following typing rules:

$$\Gamma \vdash n : \texttt{Nat} \qquad \text{(T-NAT)}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma,\, x : T_1 \vdash t : T_2}{\Gamma \vdash (\lambda x{:}T_1 \,.\, t) : T_1 \to T_2} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_2 \to T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1\ t_2 : T_1} \qquad \text{(T-APP)}$$

$$\frac{\Gamma,\, X \vdash t : T}{\Gamma \vdash (\Lambda X \,.\, t) : \forall X \,.\, T} \qquad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash t_1 : \forall X \,.\, T_1}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_1} \qquad \text{(T-TAPP)}$$

Using the above typing rules, formally prove that

$$(\Lambda X \,.\, \lambda x{:}X \,.\, x)\ [\texttt{Nat}]\ 7$$

is well-typed and what its type is in the empty context $\emptyset$. (12)

Answer:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x : X \in (\emptyset,\, X,\, x : X)}{\emptyset,\, X,\, x : X \vdash x : X}\ \text{T-VAR}}{\emptyset,\, X \vdash (\lambda x{:}X\ .\ x) : X \to X}\ \text{T-ABS}}{\emptyset \vdash (\Lambda X\ .\ \lambda x{:}X\ .\ x) : \forall X\ .\ X \to X}\ \text{T-TABS}}{\emptyset \vdash (\Lambda X\ .\ \lambda x{:}X\ .\ x)\ [\texttt{Nat}] : [X \mapsto \texttt{Nat}](X \to X)}\ \text{T-TAPP} \qquad \dfrac{}{\emptyset \vdash 7 : \texttt{Nat}}\ \text{T-NAT}}{\emptyset \vdash (\Lambda X\ .\ \lambda x{:}X\ .\ x)\ [\texttt{Nat}]\ 7 : \texttt{Nat}}\ \text{T-APP}$$

(b) Consider the following command language fragment:

$$\begin{array}{lcll}
c & \to & & \textit{commands:} \\
 & | & \texttt{skip} & \textit{no operation} \\
 & | & c\ \texttt{;}\ c & \textit{sequence} \\
 & | & \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c & \textit{conditional} \\
 & | & \texttt{while}\ e\ \texttt{do}\ c & \textit{iteration} \\
 & | & \texttt{break}\ n & \textit{break iteration} \\
 & | & \texttt{continue}\ n & \textit{continue iteration}
\end{array}$$

Here, n is the syntactic category of natural numbers, and e is the syntactic category of expressions. The details of the expression syntax is of no concern to us, but we assume a typing relation:

$$\Gamma \vdash e : T$$

meaning "expression e has type T in type context Γ". The command `break` n, where $n \geq 1$, exits from an enclosing loop, with n specifying the nesting level of the loop to exit, 1 being the innermost one. The command `continue` n transfers control to the beginning of an enclosing loop, with $n \geq 1$ again specifying the level of the loop.

It is an error to specify more enclosing loops than there are; e.g., the following is an ill-formed program fragment:

$$\texttt{while true do break}\ 2$$

Define a typing and well-formedness relation for commands that can be used to statically ensure that commands are both well-typed (here meaning that the conditions of the `if`-construct and `while`-loop both have type `Bool`; commands have no type as such) and well-formed in that the numeric argument of a `break` or `continue` command is no higher than the number of enclosing loops. Additionally, `break` 0 and `continue` 0 should be considered ill-formed, enforcing that the numeric argument to these commands is at least 1. Explain your construction and define the relation by providing an appropriate rule for each of the commands. (13)

Answer: *Define a relation:*

$$\Gamma\ ;\ n \vdash c$$

meaning "command c *is well-typed and well-formed in type context* Γ *and with* n *enclosing loops". This relation is defined by the following rules:*

$$\Gamma\ ;\ n \vdash \texttt{skip} \qquad \text{(T-SKIP)}$$

$$\frac{\Gamma\ ;\ n \vdash c_1 \quad \Gamma\ ;\ n \vdash c_2}{\Gamma\ ;\ n \vdash c_1\ ;\ c_2} \qquad \text{(T-SEQ)}$$

$$\frac{\Gamma \vdash e : \texttt{Bool} \quad \Gamma\ ;\ n \vdash c_1 \quad \Gamma\ ;\ n \vdash c_2}{\Gamma\ ;\ n \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2} \qquad \text{(T-COND)}$$

$$\frac{\Gamma \vdash e : \texttt{Bool} \quad \Gamma\ ;\ (n+1) \vdash c_1}{\Gamma\ ;\ n \vdash \texttt{while}\ e\ \texttt{do}\ c} \qquad \text{(T-WHILE)}$$

$$\frac{1 \leq m \leq n}{\Gamma\ ;\ n \vdash \texttt{break}\ m} \qquad \text{(T-BREAK)}$$

$$\frac{1 \leq m \leq n}{\Gamma\ ;\ n \vdash \texttt{continue}\ m} \qquad \text{(T-CONT)}$$

The idea is simple: we just keep a count n *of the number of enclosing loops and, whenever we encounter a* ***`break`*** m *or* ***`continue`*** m *we ensure that* m *is in the appropriate range:* $1 \leq m \leq n$.

Question 6

(a) Consider the following simple expression language, where x is the syntactic category of variables:

e	$\longrightarrow$		*expressions:*
		x	*variable*
	$\mid$	n	*constant number,* $n \in \mathbb{N}$
	$\mid$	`true`	*constant true*
	$\mid$	`false`	*constant false*
	$\mid$	`not` e	*logical negation*
	$\mid$	e `&&` e	*logical conjunction*
	$\mid$	e `+` e	*addition*
	$\mid$	e `-` e	*subtraction*
	$\mid$	e `=` e	*numeric equality test*
	$\mid$	e `<` e	*numeric less than test*

Evaluation of expressions in this language has *no side effects*. We wish to give a *denotational semantics* for this language. Assume that we take the semantic domain to be $\mathbb{N}$, the natural numbers, letting the denotation of `true` be 1 and the denotation of `false` be 0. The result of subtracting a larger number from a smaller one should be 0. Note that this expression language thus is *total*: an expression always has a well-defined result. Assume further that the meaning of a variable is obtained by looking it up in a store σ represented by a function of type Σ mapping a variable name (in the syntactic category x) to its current value (a natural number in $\mathbb{N}$):

$$\begin{array}{rcl} \Sigma & = & x \rightarrow \mathbb{N} \\ \sigma & : & \Sigma \end{array}$$

Given this, suggest an appropriate type signature for a semantic function $\mathrm{E}[\![\cdot]\!]$ that maps an expression to its meaning. Then define $\mathrm{E}[\![\cdot]\!]$ for the above expression language. You only have to consider the cases for *variable*, *constant number*, *logical conjunction*, *subtraction*, and *numeric equality test.*

(7)

Answer:

An appropriate type signature for $\mathrm{E}[\![\cdot]\!]$ *is:*

$$\mathrm{E}[\![\cdot]\!] : e \rightarrow (\Sigma \rightarrow \mathbb{N})$$

Definition (all cases given for reference):

$$
\begin{array}{rcl}
\mathrm{E}[\![x]\!]\ \sigma & = & \sigma\ x \\
\mathrm{E}[\![n]\!]\ \sigma & = & n \\
\mathrm{E}[\![\texttt{true}]\!]\ \sigma & = & 1 \\
\mathrm{E}[\![\texttt{false}]\!]\ \sigma & = & 0 \\
\mathrm{E}[\![\texttt{not}\ e]\!]\ \sigma & = & \begin{cases} 1, & \text{if } \mathrm{E}[\![e]\!]\ \sigma = 0 \\ 0, & \text{otherwise} \end{cases} \\
\mathrm{E}[\![e_1\ \texttt{\&\&}\ e_2]\!]\ \sigma & = & \begin{cases} 1, & \text{if } \mathrm{E}[\![e_1]\!]\ \sigma = 1\ \wedge\ \mathrm{E}[\![e_2]\!]\ \sigma = 1 \\ 0, & \text{otherwise} \end{cases} \\
\mathrm{E}[\![e_1\ \texttt{+}\ e_2]\!]\ \sigma & = & \mathrm{E}[\![e_1]\!]\ \sigma\ +\ \mathrm{E}[\![e_2]\!]\ \sigma \\
\mathrm{E}[\![e_1\ \texttt{-}\ e_2]\!]\ \sigma & = & \max(\mathrm{E}[\![e_1]\!]\ \sigma\ -\ \mathrm{E}[\![e_2]\!]\ \sigma,\ 0) \\
\mathrm{E}[\![e_1\ \texttt{=}\ e_2]\!]\ \sigma & = & \begin{cases} 1, & \text{if } \mathrm{E}[\![e_1]\!]\ \sigma\ =\ \mathrm{E}[\![e_2]\!]\ \sigma \\ 0, & \text{otherwise} \end{cases} \\
\mathrm{E}[\![e_1\ \texttt{<}\ e_2]\!]\ \sigma & = & \begin{cases} 1, & \text{if } \mathrm{E}[\![e_1]\!]\ \sigma\ <\ \mathrm{E}[\![e_2]\!]\ \sigma \\ 0, & \text{otherwise} \end{cases}
\end{array}
$$

(b) Now consider extending the expression language above with commands as follows. Unlike expressions, execution of a command in general has a *side effect*; that is, it may change the store:

$$
\begin{array}{rcll}
c & \rightarrow & & \textit{commands:} \\
 & | & \texttt{skip} & \textit{no operation} \\
 & | & x\ \texttt{:=}\ e & \textit{assignment} \\
 & | & c\ \texttt{;}\ c & \textit{sequence} \\
 & | & \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c & \textit{conditional} \\
 & | & \texttt{repeat}\ c\ \texttt{until}\ e & \textit{iteration}
\end{array}
$$

What is a suitable type signature for a semantic function $\mathrm{C}[\![\cdot]\!]$ mapping a command to its meaning? Provide a brief explanation of your answer. (4)

Answer: *A suitable type signature is:*

$$\mathrm{C}[\![\cdot]\!] : c \rightarrow (\Sigma \rightarrow \Sigma_\perp)$$

That is, a command is mapped to a state transformer, a function mapping a state represented by a store to a new state. However, as the commands include a loop, the execution may diverge (fail to terminate). The range (co-domain) of the state transformer must therefore be extended to include $\perp$ denoting divergence, hence $\Sigma_\perp$.

(c) Define $\mathrm{C}[\![\cdot]\!]$ for the commands given above. Use the notation $[x \mapsto v]\sigma$ to denote an updated store that is like σ except that x is mapped to the value v (in $\mathbb{N}$). The `repeat`-loop should have the usual semantics; i.e., repetition of the loop body at least once until the condition becomes true. (8)

Answer:

$$\begin{array}{rcl}
\mathrm{C}[\![\texttt{skip}]\!]\ \sigma & = & \sigma \\
\mathrm{C}[\![x\ \texttt{:=}\ e]\!]\ \sigma & = & [x \mapsto \mathrm{E}[\![e]\!]\ \sigma]\sigma \\
\mathrm{C}[\![c_1\ \texttt{;}\ c_2]\!]\ \sigma & = & (\mathrm{C}[\![c_2]\!]_{\perp\!\!\!\perp})\ (\mathrm{C}[\![c_1]\!]\ \sigma) \\
\mathrm{C}[\![\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!]\ \sigma & = & \begin{cases} \mathrm{C}[\![c_1]\!]\ \sigma, & \text{if } \mathrm{E}[\![e]\!]\ \sigma = 1 \\ \mathrm{C}[\![c_2]\!]\ \sigma, & \text{otherwise} \end{cases} \\
\mathrm{C}[\![\texttt{repeat}\ c\ \texttt{until}\ e]\!] & = & \mathrm{fix}_{\Sigma \to \Sigma_\perp}\left(\lambda f.\lambda\sigma.\begin{cases} \sigma', & \text{if } \mathrm{E}[\![e]\!]_{\perp\!\!\!\perp}\ \sigma' = 1 \\ f_{\perp\!\!\!\perp}\ \sigma', & \text{otherwise} \\ \text{where } \sigma' = \mathrm{C}[\![c]\!]\ \sigma \end{cases}\right)
\end{array}$$

(d) Suppose we wish to add expressions *with* side effects to the language; for example, a C-like post-increment operator: `x++`. Explain how the semantics would have to be restructured. Assume that the expression sublanguage still is total and terminating. Illustrate your answer by giving suitable definitions for $\mathrm{E}[\![e_1\ \texttt{+}\ e_2]\!]$ and $\mathrm{C}[\![x\ \texttt{:=}\ e]\!]$.

(6)

Answer: *As the evaluation of an expression now can have a side effect, the semantic function* $\mathrm{E}[\![\cdot]\!]$ *must be changed to return a possibly changed store in addition to the result of the expression. As the expression sublanguage still is total and terminating, we do not need to lift the range of the function. The new type signature for* $\mathrm{E}[\![\cdot]\!]$ *thus becomes:*

$$\mathrm{E}[\![\cdot]\!] : e \to (\Sigma \to \mathbb{N} \times \Sigma)$$

The possibly changed store has to be threaded properly through the evaluation of subexpressions, as the case for evaluation of e.g. addition illustrates:

$$\begin{array}{rcl}
\mathrm{E}[\![e_1\ \texttt{+}\ e_2]\!]\ \sigma & = & (n_1 + n_2,\ \sigma'') \\
& & \text{where } \begin{array}{rcl} (n_1,\ \sigma') & = & \mathrm{E}[\![e_1]\!]\ \sigma \\ (n_2,\ \sigma'') & = & \mathrm{E}[\![e_2]\!]\ \sigma' \end{array}
\end{array}$$

Whenever the execution of a command involves the evaluation of an expression, the possibly changed store again has to be threaded through properly as the case for assignment illustrates:

$$\begin{array}{rcl}
\mathrm{C}[\![x\ \texttt{:=}\ e]\!]\ \sigma & = & [x \mapsto n]\sigma' \\
& & \text{where } (n,\ \sigma') = \mathrm{E}[\![e]\!]\ \sigma
\end{array}$$

Appendix A

This appendix contains the abstract syntax, operational semantics, and typing rules for the small example language (from Pierce's book *Types and Programming Languages*) that has been discussed in the module.

Abstract Syntax:

t	$\rightarrow$			*terms:*
			`true`	*constant true*
		$\mid$	`false`	*constant false*
		$\mid$	`if` t `then` t `else` t	*conditional*
		$\mid$	`0`	*constant zero*
		$\mid$	`succ` t	*successor*
		$\mid$	`pred` t	*predecessor*
		$\mid$	`iszero` t	*zero test*

Values:

v	$\rightarrow$			*values:*
			`true`	*true value*
		$\mid$	`false`	*false value*
		$\mid$	nv	*numeric value*
nv	$\rightarrow$			*numeric values:*
			`0`	*zero value*
		$\mid$	`succ` nv	*successor value*

Types:

T	$\rightarrow$			*types:*
			`Bool`	*Boolean type*
		$\mid$	`Nat`	*Numeric type*

Operational Semantics: (call-by-value)

$$\texttt{if true then } t_2 \texttt{ else } t_3 \longrightarrow t_2 \qquad \text{(E-IFTRUE)}$$

$$\texttt{if false then } t_2 \texttt{ else } t_3 \longrightarrow t_3 \qquad \text{(E-IFFALSE)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3} \qquad \text{(E-IF)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{succ } t_1 \longrightarrow \texttt{succ } t_1'} \qquad \text{(E-SUCC)}$$

$$\texttt{pred 0} \longrightarrow \texttt{0} \qquad \text{(E-PREDZERO)}$$

$$\texttt{pred } (\texttt{succ } nv_1) \longrightarrow nv_1 \qquad \text{(E-PREDSUCC)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{pred } t_1 \longrightarrow \texttt{pred } t_1'} \qquad \text{(E-PRED)}$$

$$\texttt{iszero 0} \longrightarrow \texttt{true} \qquad \text{(E-ISZEROZERO)}$$

$$\texttt{iszero } (\texttt{succ } nv_1) \longrightarrow \texttt{false} \qquad \text{(E-ISZEROSUCC)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{iszero } t_1 \longrightarrow \texttt{iszero } t_1'} \qquad \text{(E-ISZERO)}$$

Typing Rules:

$$\texttt{true} : \texttt{Bool} \qquad \text{(T-TRUE)}$$

$$\texttt{false} : \texttt{Bool} \qquad \text{(T-FALSE)}$$

$$\frac{t_1 : \texttt{Bool} \quad t_2 : T \quad t_3 : T}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T} \qquad \text{(T-IF)}$$

$$\texttt{0} : \texttt{Nat} \qquad \text{(T-ZERO)}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{succ } t_1 : \texttt{Nat}} \qquad \text{(T-SUCC)}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{pred } t_1 : \texttt{Nat}} \qquad \text{(T-PRED)}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{iszero } t_1 : \texttt{Bool}} \qquad \text{(T-ISZERO)}$$

End