Exam in TDT4165 Programming Languages: Solutions

Author: Wacław Kuśnierczyk*

Exam Date: August 8., 2007

Solutions

Problem 1 (15%)

Task 1a

Answer. According to the pensum book (p. 178), *procedural abstraction* is the ability to convert any statement into a procedure value. That is, a piece of code that contains a statement or a sequence of statements:

$\langle$*statement*$\rangle_1$
$\langle$*statement*$\rangle_2$
...
$\langle$*statement*$\rangle_n$

can be replaced by an application of an appropriately defined procedure having those statements as its body:

`proc {`$\langle$*procedure*$\rangle$`}` $\langle$*statement*$\rangle_1$ $\langle$*statement*$\rangle_2$ ... $\langle$*statement*$\rangle_n$ `end`
`{`$\langle$*procedure*$\rangle$`}`

There are a number of benefits of using procedural abstraction. For example,

- if the same statement or sequence of statements appears in more than one place in the code, the program may be made cleaner if an appropriate procedure is defined once and the statements are then replaced by procedure calls;
- the procedure can be parameterized (i.e., it may take arguments), so that a number of similar statements that differ in some detail can be replaced by an application of the same procedure, with different arguments;
- the definition of a procedure and its execution (an application) do not have to appear in the same place; thus, statements can be introduced in one place (e.g., in the header of a program), but executed elsewhere;

  etc.

*Contact info: waku@idi.ntnu.no

Task 1b

Answer. `FoldRight` and `FoldLeft` can be implemented as follows:

```
fun {FoldRight List Combine Transform Null}
   case List
   of nil then Null
   [] Head|Tail then
      {Combine {Transform Head} {FoldRight Tail Combine Transform Null}}
   end
end

fun {FoldLeft List Combine Transform Null}
   case List
   of nil then Null
   [] Head|Tail then
      {FoldLeft Tail Combine Transform {Combine Null {Transform Head}}}
   end
end
```

Both definitions are recursive: both `FoldRight` and `FoldLeft` call themselves in their bodies, rather than use a looping construct such as `for ... end`. Execution of `FoldRight` lead to a recursive process (the operation of combination can be performed only after the rest of the list have been folded, i.e., it must wait until the subsequent recursive calls return). Execution of `FoldLeft` lead to an iterative process (the operation of combination is performed first, and then folding is performed tail-recursively, i.e., there is no operation waiting for the result of the subsequent recursive call).

Task 1c

Answer. `Map` can be implemented using both the `FoldRight` and `FoldLeft` defined above, as follows:

```
fun {MapRight List Function}
   {FoldRight List
    fun {$ X Y} X|Y end
    fun {$ X} {Function X} end
    nil}
end
fun {MapLeft List Function}
   {FoldLeft {Reverse List}
    fun {$ X Y} Y|X end
    fun {$ X} {Function X} end
    nil}
end
```

`MapLeft` is less effective, since it requires the input list to be reversed first (thus, the list is traversed twice rather than once, as in the case of `MapRight`).

If you answered that `FoldLeft` cannot be used to implement `Map`, and you provided a good explanation, the answer was considered correct.

Problem 2 (20%)

Task 2a

Answer. Acording to the pensum book (p. 420), a bundled data abstraction is a data abstraction that defines just one type of entity: an object containing both data (values) and methods (operations). An unbundled data abstraction (an abstract data type) is a data abstraction which defines two separate kinds of entities, values and operations.

In the case of `NewStructure`, we do not see the definition, and we do not know whether it is part of an implementation of a bundled or unbundled data abstraction. While there are procedures (functions) external to `Structure` applied to it, it does not prove that the data abstraction is unbundled. It is possible to define functions external to an object which simply call the internal procedures of the object — see below.

Unbundled data abstractions are widely used in programming. Object-oriented programming (OOP) languages are based on bundled abstractions, but many languages do not support OOP, or support both OOP and unbundled abstract data types.

Task 2b

Answer. According to the pensum book (p. 420), a data abstraction is secure if its encapsulation is enforced by the language, rather than explicitly by the programmer who uses the data abstraction. The content of a secure data structure can be accessed only through the operations defined by the data asbtraction, while the content of a non-secure (open) data structure can be accessed and modified by the programmer without using those operations.

Again, we do not know whether structures produced by `NewStructure` are secure or not. It is possible that such structures are bundled but insecure (they expose both the encapsulated operations and values), bundled and secure (expose only the operations), unbundled and secure (the operations `Get` and `Put` must know a secret key to access the content of the structure), or unbundled and insecure (the content is accessible to anyone).

It is possible to implement a secure data abstraction in a declarative model of computation by means of encapsulation as a bundled abstraction, a declarative object. It is also possible to implement an unbundled secure data abstraction in a declarative model, though without the use of unique names (which are available only in a non-declarative model) it is not particularly useful (each instance has to have a key explicitly defined by the programmer, rather than generated by `NewName`).

Declarative data abstractions can be defined in a non-declarative model of computation. If the abstractions do not use explicit state, they will behave declaratively, even if the rest of the program performs stateful computations.

Non-secure (open) data abstractions are widely used in programming, and are useful especially in situations where security is not essential and easy access to the content of a structure is more inmportant.[1]

Task 2c

Answer. Examples of linear data abstractions are stacks (LIFO, last in first out linear structures), queues (FIFO, first in first out linear structures), arrays or vectors (random-access linear struc-

[1] Strictly speaking, if it is possible to access the content of a data structure omitting the allowed operations, the data structure is not an instance of a data abstraction.

tures), etc. Examples of non-linear data abstractions are trees, heaps, directed acyclic graphs, etc.

The data abstraction represented by `NewStructure` can be linear or non-linear. We do not see its definition, but from the behaviour and the names of the operations used, it is reasonable to assume that it is a stack.

Task 2d

Answer. `NewStructure` and the procedures `Put` and `get` can be implemented as follows:

```
NewStructure =
local
   fun {Pack Content}
      fun {Put Item}
         {Pack Item|Content}
      end
      fun {Get Item}
         case Content
         of Head|Tail then Item = Head {Pack Tail} end
      end
   in
      structure(put:Put get:Get)
   end
in
   fun {$} {Pack nil} end
end


fun {Put Structure Item} {Structure.put Item} end

fun {Get Structure Item} {Structure.get Item} end
```

The abstraction is declarative: there is no mutable state kept inside `Structure`; any operation that changes the content must result in a new structure, rather than in an update of the existing one. The abstraction is bundled: the operations are encapsulated within an object; the external `Put` and `get` are mere syntactic sugar, and they call the internal operations of the object passed to them as an argument. The abstraction is secure: the content of a structure can be accessed only through the operations encapsulated in it.

Task 2e

Answer. Given the three orthogonal (independent) binary criteria — declarativeness, bundledness, openness — we can think of eight different types of data abstractions. The pensum book discusses the utility of five of those, and illustrates the discussion with implementations of the stack data abstraction:

1. an open, declarative, unbundled stack;
2. a secure, declarative, unbundled stack;
3. a secure, declarative, bundled stack;

4. a secure, stateful (non-declarative), unbundled stack;

5. a secure, stateful, bundled stack.

For details, see p. 420. In principle, it is possible to implement data abstractions of the remaining three types, but this is not necessarily very useful (once they are implemented, it is easy to turn them into secure ones).

Of the five types above, only the first and the third can be implemented in a declarative model of computation. The second type requires unique names, which are available only in a non-declarative model (even if the data abstraction is declarative), and the remaining two require explicit state.

None of these types requires concurrency, and thus none of them requires shared-state concurrency.

Problem 3 (20%)

Task 3a

Answer. Lazy execution is a strategy for the execution of statements in which statements are executed when the results of the execution are needed, not when the statements are met in the program. Lazy execution is useful, for example, to postpone the computation of an infinite stream by a producer until elements of the stream are demanded by consumers, i.e., to implement demand-driven concurrent computation.

Task 3b

Answer. `SumSquares` can be implemented in the kernel language as follows:

```
SumSquares =
proc {$ N1 N2 ?Result}
   local Compute in
      Compute =
      proc {$ ?Result}
         Result = N1*N1+N2*N2
      end
      {ByNeed Compute Result}
   end
end
```

Task 3c

Answer. `ListEvenIntegers` and the helper function `ListItem` can be implemented as follows:[2]

```
fun {ListEvenIntegers}
   fun lazy {Enumerate N}
      N|{Enumerate N+2}
   end
in
```

[2] Of course, not in the kernel language.

```
   {Enumerate 0}
end

fun {ListItem List N}
   case List
   of Head|Tail then
      if N == 1 then Head
      else {ListItem Tail N-1}
      end
   end
end
```

Task 3d

Answer. The abstract machine for the kernel language must include a trigger store. The semantic statement ({`ByNeed` $\langle x \rangle$ $\langle y \rangle$}, E) is executed in the following steps (p. 282 in the pensum book):

- if E($\langle x \rangle$) is determined, a new thread with the semantic statement ({$\langle x \rangle$ $\langle y \rangle$}, E) is created;
- if E($\langle x \rangle$) is not determined, the pair (E($\langle x \rangle$), E($\langle y \rangle$)) is added to the trigger store (unless there already is such a pair there).

An application of a lazy function is equivalent to a by-need application of a one-argument procedure x to an unbound variable y which will be bound by the procedure as a result of the application, when needed (as in Task 3b). When the variable is needed, the pair (x, y) is removed from the trigger store, and a thread is created with the semantic statement ({$\langle x \rangle$ $\langle y \rangle$}, {$\langle x \rangle \rightarrow$ x, $\langle y \rangle \rightarrow$ y}), where $\langle x \rangle$ and $\langle y \rangle$ are any two distinct identifiers. A variable is needed if its value must be determined in order for some operation to continue. If the variable is bound, its value is retrieved from the single assignment store. If the variable is not bound, and it appears as the second element of some pair in the trigger store, a new thread is created, as above. If the variable is not bound, but there is no corresponding trigger, the operation which needs the variable suspends. The first line in the following code specifies an operation in which the variable named by the identifier `Variable` is needed, the second line — an operation where the variable is not needed:

```
{Browse Variable+0}
{Browse Variable}
```

Lazy execution, as specified above, requires the following riles for variable reachability:

- if a variable appears as the first element of a pair in the trigger store, and the second element of that pair is reachable, then the variable is reachable;
- if a variable is unreachable, then every pair in the trigger store that contains the variable as the second element should be removed from the trigger store.

Task 3e

Answer. The first line results in the creation of a new thread which, when activated by the scheduler, will apply `Procedure` to `Argument`.

The second line results in the creation of a new thread (if `Argument` is determined), or in the creation of a trigger (if `Argument` is not determined). In the latter case, no new thread is created until there is a need for the value of `Argument`.

Problem 4 (15%)

Task 4a

Answer. The grammar for σ can be defined as follows:

$$
\begin{array}{rcl}
\langle \textit{program} \rangle & ::= & \{\langle \textit{instruction} \rangle\}^{+} \\
\langle \textit{instruction} \rangle & ::= & \langle \textit{definition} \rangle \mid \langle \textit{expression} \rangle \\
\langle \textit{definition} \rangle & ::= & \langle \textit{variable-definition} \rangle \mid \langle \textit{function-definition} \rangle \\
\langle \textit{expression} \rangle & ::= & \langle \textit{identifier} \rangle \mid \langle \textit{numeral} \rangle \mid \langle \textit{application} \rangle \\
\langle \textit{variable-definition} \rangle & ::= & \text{`('} \ \texttt{define} \ \text{'} \langle \textit{identifier} \rangle \ \langle \textit{expression} \rangle \ \text{`)'} \\
\langle \textit{function-definition} \rangle & ::= & \text{`('} \ \texttt{define} \ \text{'} \text{`('} \ \langle \textit{identifier} \rangle \ \langle \textit{identifier} \rangle \ \text{`)'} \ \langle \textit{expression} \rangle \ \text{`)'} \\
\langle \textit{application} \rangle & ::= & \text{`('} \ \langle \textit{identifier} \rangle \ \langle \textit{expression} \rangle \ \text{`)'}
\end{array}
$$

Task 4b

Answer. The grammar for σ is a context-free grammar: one in which each rule is of the form $A ::= \gamma$, where γ is a string of terminals and nonterminals. A context-sensitive grammar is one in which the rules have the form $\alpha A \beta ::= \alpha \gamma \beta$, with α and β strings of terminals and nonterminals. Since both α and β may be empty, every context-free grammar (such as the one for σ) is also a context-sensitive grammar — thus, the grammar above *is* context-sensitive.[3] The following is an example of a non-context-free grammar:

$$
\begin{array}{rcl}
\langle \textit{operator} \rangle & ::= & \text{`+'} \mid \text{`-'} \mid \text{`*'} \mid \text{`/'} \\
\langle \textit{operator} \rangle \langle \textit{numeral} \rangle & ::= & \text{`-'} \ \langle \textit{numeral} \rangle
\end{array}
$$

The grammar for σ is not ambiguous. A grammar is ambiguous if there are strings that can be generated by the grammar in more than one way — strings that have more than one parse tree, or more than one leftmost derivation. The following is a simple example of an ambiguous grammar:

$$
\langle \textit{expression} \rangle \quad ::= \quad \langle \textit{expression} \rangle \ \text{`-'} \ \langle \textit{expression} \rangle \mid \text{`1'}
$$

The string `1 - 1 - 1`, for example, has two parse trees.

Task 4c

Answer. The syntax-checker can be implemented as follows:

```
\insert solve.oz

declare

fun {CheckSyntax Tokens}
   {SolveAll fun {$} {CheckProgram Tokens nil} true end} \= nil
end

proc {CheckProgram Tokens Rest}
   Instructions in
   choice
```

[3] If you answered that the grammar for σ is not context-sensitive because it is context-free, the answer was not considered incorrect, though, strictly speaking, it is incorrect.

```
      {CheckExpression Tokens Instructions}
   [] {CheckDefinition Tokens Instructions}
   end
   {CheckInstructions Instructions Rest}
end

proc {CheckInstructions Tokens Rest}
   choice
      Tokens = nil Rest = nil
   [] {CheckProgram Tokens Rest}
   end
end

proc {CheckDefinition Tokens Rest}
   choice
      {CheckVariableDefinition Tokens Rest}
   [] {CheckFunctionDefinition Tokens Rest}
   end
end

proc {CheckExpression Tokens Rest}
   choice
      {CheckIdentifier Tokens Rest}
   [] {CheckNumber Tokens Rest}
   [] {CheckApplication Tokens Rest}
   end
end

proc {CheckVariableDefinition Tokens Rest}
   Identifier Expression in
   Tokens = '('|'define'|Identifier
   {CheckIdentifier Identifier Expression}
   {CheckExpression Expression ')'|Rest}
end

proc {CheckFunctionDefinition Tokens Rest}
   FunctionIdentifier ArgumentIdentifier Expression in
   Tokens = '('|'define'|'('|FunctionIdentifier
   {CheckIdentifier FunctionIdentifier ArgumentIdentifier}
   {CheckIdentifier ArgumentIdentifier ')'|Expression}
   {CheckExpression Expression ')'|Rest}
end

proc {CheckIdentifier Tokens Rest}
   Tokens = identifier(_)|Rest
end

proc {CheckNumber Tokens Rest}
   Tokens = number(_)|Rest
end

proc {CheckApplication Tokens Rest}
   Identifier Expression in
   Tokens = '('|Identifier
   {CheckIdentifier Identifier Expression}
   {CheckExpression Expression ')'|Rest}
end
```

Problem 5 (20%)

Task 5a

Answer. The data abstraction can be specified as follows:

{ d is undefined }
make d an empty dequeue
{ $d = \langle\rangle$ }

{ $d = \langle v_1, v_2, \ldots, v_n \rangle$, $e = v$ }
push the value of e onto d
{ $d = \langle v_1, v_2, \ldots, v_n, v \rangle$, $e = v$ }

{ $d = \langle v_1, v_2, \ldots, v_{n-1}, v_n \rangle$, e is undefined }
pop from d a value and store it in e
{ $d = \langle v_1, v_2, \ldots, v_{n-1} \rangle$, $e = v_n$ }

{ $d = \langle v_1, v_2, \ldots, v_n \rangle$, $e = v$ }
shift the value of e onto d
{ $d = \langle v, v_1, v_2, \ldots, v_n \rangle$, $e = v$ }

{ $d = \langle v_1, v_2, \ldots, v_{n-1}, v_n \rangle$, e is undefined }
unshift from d a value and store it in e
{ $d = \langle v_2, \ldots, v_n \rangle$, $e = v_1$ }

Task 5b

Answer. The function `NewDequeue` can be implemented as follows:

```
fun {NewDequeue}
   Content = {NewCell nil}
   proc {Push Item}
      Content := {List.append @Content [Item]}
   end
   proc {Pop ?Item}
      if @Content \= nil then
         Item = {List.last @Content}
         Content := {List.take @Content {List.length @Content}-1}
      end
   end
   proc {Shift Item}
      Content := Item|@Content
   end
   proc {Unshift ?Item}
      case @Content
      of Head|Tail then
         Item = Head
```

```
            Content := Tail
         end
      end
   in
      proc {$ Message}
         case Message of
            push(Item) then {Push Item}
         [] pop(Item) then {Pop Item}
         [] shift(Item) then {Shift Item}
         [] unshift(Item) then {Unshift Item}
         end
      end
   end
```

Task 5c

Answer. Dequeues as implemented above are secure: the cell `Content` is accessible only to the procedures `Push`, `Pop`, `Shift`, and `Unshift`; in turn, these procedures are accessible only through sending messages to the anonymous procedure (the dispatcher) returned by `NewDequeue`.

Problem 6 (10%)

Task 6a

Answer. The clients are implemented in a declarative way: none of the elements of their implementation requires a non-declarative model of computation. However, their behaviour is not necessarily declarative: the result of an application of a client's `compute` function is dependent on what the server does, and the server may be non-declarative, i.e., the behaviour of the clients may be dependent on an external mutable state, that of the server. With an implementation of the server as below, clients will behave declaratively, even though the server has an internal mutable state.

Both answers positive and negative answers were considered correct if sufficient justification was provided.

Task 6b

Answer. The server can be implemented as follows:

```
fun {NewServer}
   InputStream
   InputPort = {NewPort InputStream}
   proc {Process Stream}
      case Stream
      of message(Input Function Output)|Rest
      then Output = {Function Input}
         {Process Rest}
      end
   end
   proc {Call Message}
      {Send InputPort Message}
```

```
   end
in
   thread {Process InputStream} end
   server(call:Call)
end
```

The server is not a declarative object: its implementation requires ports, a feature of a non-declarative model of computation.

Task 6c

Answer. For an implementation of a server to which an arbitrary (an not specified *a priori*) number of clients can be connected at runtime, mutable state is necessary. At minimum, ports must be available. A purely declarative model of computation is insufficient.