Victoria University of Wellington

DEGREE EXAMINATIONS — 1998

MID-YEAR

COMP 426

Formal Software Development

Time Allowed: 3 Hours

Instructions: Candidates should attempt **all** questions.

This exam will be marked out of 100.

A summary of Z mathematical notation is provided.

Questions **1** to **4** relate to the following specification.

A *storage manager* manages the allocation of storage space among client *processes*, by allocating blocks of addresses to processes and reclaiming addresses that are no longer needed.

The store is modelled as a vector (i.e. a fixed length sequence) of *StoreSize* storage units, which are indexed by *addresses* (*ADDR*). A *block* is a non-empty set of contiguous addresses. Each process is identified by a *process identifier* (*PID*).

$[PID]$

$| \; StoreSize : \mathbb{N}_1$

$$ADDR == 1 \mathrel{..} StoreSize$$
$$BLOCK == \{\, b : \mathbb{P}_1\, ADDR \mid (\forall\, i, j : b \bullet i \mathrel{..} j \subseteq b) \,\}$$

The storage manager records which addresses are allocated to which processes, and which are free (i.e. unallocated).

StoreMan

$alloc : ADDR \pfun PID$
$free : \mathbb{P}\, ADDR$

$free = ADDR \setminus \operatorname{dom} alloc$

Initially, no addressees are allocated (so they must all be free).

$$
\begin{array}{|l}
\textit{InitStoreMan} \\
\hline
\textit{StoreMan}' \\
\hline
\textit{alloc}' = \varnothing \\
\hline
\end{array}
$$

When a process requests a block of addresses, the storage manager finds a block of the required size, if there is one, allocates it to the requesting process, and returns lowest address in the block; otherwise the operation fails.

$$\textit{Outcome} ::= ok \mid \textit{fail}$$

The operation is specified in two parts; one where the operation succeeds (with an outcome of *ok*) and one where it fails.

$$
\begin{array}{|l}
\textit{AllocateS} \\
\hline
\Delta \textit{StoreMan} \\
p? : PID \\
n? : \mathbb{N}_1 \\
a! : ADDR \\
r! : \textit{Outcome} \\
\hline
\exists\, b : BLOCK \mid b \subseteq \textit{free} \land \# b = n? \bullet \\
\qquad \textit{free}' = \textit{free} \setminus b \land \\
\qquad \textit{alloc}' = \textit{alloc} \cup \{\ i : b \bullet i \mapsto p?\ \} \land \\
\qquad a! = \textit{min}\ b \land \\
\qquad r! = ok \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\textit{AllocateF} \\
\hline
\Xi \textit{StoreMan} \\
n? : \mathbb{N}_1 \\
r! : \textit{Outcome} \\
\hline
(\ \forall\, b : BLOCK \mid b \subseteq \textit{free} \bullet \# b < n?) \\
r! = \textit{fail} \\
\hline
\end{array}
$$

$$\textit{Allocate} == \textit{AllocateS} \lor \textit{AllocateF}$$

Question 1. **[10 marks]**

Calculate the preconditions of *AllocateS* and *AllocateF*, and use them to determine whether *Allocate* is defined for all inputs.

Question 2. **[10 marks]**

The storage manager also has a *Free* operation for freeing addresses, which is specified in two parts as in the *Allocate* operation.

Complete following schema definition, specifying the "success" case for the *Free* operation. *FreeS* should free a block of addresses, provided those addresses are all allocated to the requesting process.

$$
\begin{array}{|l}
\hline
\textit{FreeS} \\
\Delta \textit{StoreMan} \\
p? : \textit{PID} \\
a? : \textit{ADDR} \\
n? : \mathbb{N}_1 \\
r! : \textit{Outcome} \\
\hline
\ldots \\
\hline
\end{array}
$$

Question 3. **[10 marks]**

Consider the following properties, which might be desired of a storage manager:

(a) No address can be allocated to two processes at the same time.

(b) If the storage manager receives a request to allocate n words of storage, the request will always succeed if there are at least n words of storage free.

(c) A process can only free a block of storage, beginning at address a and n words long, if that storage was allocated by a call on *Allocate* with n as its size input, which returned a as its address output.

For each of these properties, say whether the property can be proved to hold for the storage manager specified above. Assume the storage manager has only the operations *InitStoreMan*, *Allocate* and *Free*, as described above.

If the property can be proved, give a *brief* sketch of the proof.

If the property cannot be proved, explain why it does not hold.

Question 4. **[20 marks]**

Now consider how you might implement the storage manager. In order to obtain an efficient implementation, we make the concept of block explicit in the representation, and record information about blocks, rather than about individual addresses.

We describe storage allocation using two partial functions, *size* and *owner*, as follows. If a is the lowest address in a block, $size(a)$ gives the size of the block, and $owner(a) = p$ if the block is currently allocated to process p.

There can be no overlapping blocks, and the collection of blocks described by *size* must account for all of $ADDR$. That is to say, every address must belong to exactly one block. To simplify this description, we first introduce a function returning the set of addresses in a block, given its start address and length.

$$extent : ADDR \times \mathbb{N} \rightarrow (\mathbb{F}\, ADDR)$$
$$\forall\, a : ADDR;\ n : \mathbb{N} \bullet$$
$$\quad extent(a, n) = a \mathrel{..} a + n - 1$$

The concrete state is now described as follows:

StoreBoy

$$size : ADDR \rightarrow\!\!\!\!\!+ \mathbb{N}_1$$
$$owner : ADDR \rightarrow\!\!\!\!\!+ PID$$
$$\mathrm{dom}\, owner \subseteq \mathrm{dom}\, size$$
$$(\forall\, a : ADDR \bullet \exists_1\, b : \mathrm{dom}\, size \bullet a \in extent(b, size(b)))$$

(a) To avoid problems later, we should also add a requirement that there can never be two adjacent free blocks.

Write an additional state invariant for *StoreBoy* imposing this condition.

(b) We can now explain how *StoreBoy* is used to represent *StoreMan*:

- *free* is the set of all addresses in unallocated blocks; i.e. the union of the extents of all unallocated blocks.
- *alloc* is a function mapping every allocated address onto its owner; i.e. the set of exactly those maplets of the form $a \mapsto p$, where a is an address in an allocated block and p is its owner.

Complete the last line of the following abstraction relation, showing how *alloc* is represented by *StoreBoy*.

Abs

StoreMan
StoreBoy

$$free = \bigcup\ \{\ a : \mathrm{dom}\, size \setminus \mathrm{dom}\, owner \bullet extent(a, size(a))\ \}$$
$$alloc = \ldots$$

(c) Show how the above abstraction relation can be used to *calculate* a data refinement of $InitStoreMan$.

You should give the general form of the formula for calculating this data refinement, and show the result of expanding the formula for this case. Simplify the resulting schema as far as you can.

Explain how the requirement that there never be two adjacent free blocks is used in simplifying the schema.

(d) The following is a specification for a concrete version of $AllocateS$:

$$
\begin{array}{|l}
\hline
AllocateS1 \\
\Delta StoreBoy \\
p? : PID \\
n? : \mathbb{N}_1 \\
a! : ADDR \\
r! : Outcome \\
\hline
a! \in \operatorname{dom} size \setminus \operatorname{dom} owner \\
n? \leq size(a!) \\
(\ n? = size(a!) \Rightarrow size' = size\ ) \\
(\ n? < size(a!) \Rightarrow size' = size \oplus \{a! \mapsto n?, a! + n? \mapsto size(a!) - n?\}\ ) \\
owner' = owner \cup \{a! \mapsto p?\} \\
r! = ok \\
\hline
\end{array}
$$

(i) Explain why two cases are required to specify the new value for $size$.

(ii) State the general forms of the proof obligations that must be proved in order to show that $AllocateS1$ is a correct data refinement of $AllocateS$. (You are not required to write out the expanded versions of the proof obligations.)
Give a *brief* explanation of why these proof obligations hold.

(iii) Is $AllocateS1$ equivalent to the specification that would be obtained by calculating the data refinement of $AllocateS$? Explain why or why not.

Question 5. **[15 marks]**

The following facts may be useful in answering this question.

A predicate transformer $wp(S)$ is:

- *conjunctive* if $wp(S, \phi \wedge \psi) \equiv wp(S, \phi) \wedge wp(S, \psi)$
- *disjunctive* if $wp(S, \phi \vee \psi) \equiv wp(S, \phi) \vee wp(S, \psi)$
- *monotonic* if for all ϕ, ψ such that $\phi \Rrightarrow \psi$, we have $wp(S, \phi) \Rrightarrow wp(S, \psi)$.

A program constructor *preserves* a property if, assuming the components have the property, one can show that the compound has the property. For example, the program constructor ';' preserves termination, because if S and T are terminating, so is $S\ ;\ T$.

A program constructor is $\sqsubseteq$-*monotonic* if refinement of components guarantees refinement of the compound. For example, ';' is $\sqsubseteq$-monotonic because if $S \sqsubseteq S'$ and $T \sqsubseteq T'$ then $S\ ;\ T \sqsubseteq S'\ ;\ T'$.

Weakest preconditions:

demonic choice	$wp(S_1 \mathbin{[\!]} S_2, \psi) = wp(S_1, \psi) \wedge wp(S_2, \psi)$
dualization	$wp(\overline{S}, \psi) = \neg wp(S, \neg\psi)$
logical constant	$wp(\lvert[\mathbf{con}\ x \bullet S]\rvert, \psi) = (\exists\, x \bullet wp(S, \psi))$, provided x does not occur in ψ

(a) Show that demonic choice

(i) preserves conjunctivity

(ii) does not preserve disjunctivity.

(b) Explain the role of **con** in formalizing initial variables.

(c) Show that **con**

(i) doesn't preserve conjunctivity in general

(ii) preserves conjunctivity when its use is restricted to the formalization of initial variables.

(d) Show that dualization preserves monotonicity.

(e) Why is $\sqsubseteq$-monotonicity an important property of program constructors?

(f) Show that

(i) demonic choice is $\sqsubseteq$-monotonic

(ii) dualization is not $\sqsubseteq$-monotonic.

Question 6. **[15 marks]**

You have an ordered sequence $A[1..n]$ of integers, and must find the position i of a given value x in it. If x does not occur in A, i should index the least element of A that is greater than x, or the position $n+1$ if there is no such element.

This problem can be specified as follows:

$$\begin{array}{l} \mathbf{var}\ A : \mathrm{seq}\,\mathbb{Z};\ x : \mathbb{Z};\ i : \mathbb{N} \bullet \\ \quad i{:}[\mathrm{true}\ /\ \mathit{ordered}(A)\ /\ A[1 \mathinner{..} i-1] < x \wedge x \leq A[i \mathinner{..} n]] \end{array}$$

If $j \leq k$, $A[j \mathinner{..} k]$ denotes the subsequence of A starting at $A[j]$ and ending at $A[k]$ (i.e., the bounds are included). If $j > k$, $A[j \mathinner{..} k]$ denotes the empty sequence.

For sequences s and t, and value x, the notation $s < x$ denotes $(\forall j : \mathrm{dom}\, s \bullet s[j] < x)$, and the notation $x \leq t$ denotes $(\forall j : \mathrm{dom}\, t \bullet x \leq t[j])$.

Since the input sequence is ordered, a binary search could be used. However, this question asks you to derive a linear search algorithm.

(a) State a suitable loop invariant. Recall that a loop invariant can be obtained by weakening the postcondition, so that the invariant is easily established initially, and the postcondition is guaranteed once the loop terminates.

(b) State a suitable variant. Recall that the variant is a decreasing integer-valued function, chosen so that the loop will terminate *no later than* the iteration where the variant reaches zero.

(c) Complete the derivation to code. For each step, state and briefly justify any proof obligations.

(d) Briefly explain how you would modify the derivation to yield a binary search.

Question 7. **[5 marks]**

One definition of data refinement is

$$S_A \preceq_{\mathit{rep}} S_C \quad \text{iff} \quad \mathit{rep}\ ;\ S_A \sqsubseteq S_C\ ;\ \mathit{rep}$$

where S_A is the abstract program, S_C is the concrete program, and *rep* is a predicate transformer corresponding to the abstraction relation.

(a) Draw a commuting diagram depicting this definition, clearly labelling each node and edge.

(b) Let *rep* be the predicate transformer that turns a postcondition ψ into the precondition $(\exists\, a \bullet I \wedge \psi)$, where a is the abstract variables in S_A and I is the coupling invariant.

Derive from this the definition of data refinement discussed in lectures, which involves *wp* directly.

Question 8. **[15 marks]**

The following module specifies a mechanism for allocating unique natural number tags, with operations (procedures) to acquire and release tags. A call on *Acquire* will yield a tag that is different from any tag that has been previously allocated but not released; a call on *Release* will release the argument tag, so it is available for reallocation.

$$
\begin{array}{ll}
\textbf{module}\ \text{Tag} & \\
\quad \textbf{var}\ \mathit{used} : \mathbb{P}\,\mathbb{N}; & \\
\quad \textbf{procedure}\ \mathit{Acquire}\ (\textbf{result}\ t : \mathbb{N})\ \widehat{=} & \\
\qquad t :\in \mathbb{N} \setminus \mathit{used}; & (i) \\
\qquad \mathit{used} := \mathit{used} \cup \{t\} & (ii) \\
\quad \textbf{procedure}\ \mathit{Release}\ (\textbf{value}\ t : \mathbb{N})\ \widehat{=} & \\
\qquad \mathit{used} := \mathit{used} \setminus \{t\} & (iii) \\
\quad \textbf{initially}\ \mathit{used} = \{\} & (iv)
\end{array}
$$

An implementation of this module can be obtained by representing the set variable *used* by an integer variable *last*, which records the value of the largest tag that has been allocated. Tags are then acquired by simply returning the next largest natural number after *last*. Released tags are never reused.

(a) Give a coupling invariant for this data refinement.

(b) Express the assignments (labelled (i)–(iii)) as specification statements.

(c) Show the results of augmenting the commands (i)–(iii) and the initialization (iv) with the new concrete variable *last*, using the coupling invariant from part (a).

(d) State and prove the proof obligations for transforming the results of part (c) to give the following augmented procedure definitions:

$$
\begin{array}{l}
\textbf{procedure}\ \mathit{Acquire}\ (\textbf{result}\ t : \mathbb{N})\ \widehat{=} \\
\quad t := \mathit{last} + 1; \\
\quad \mathit{last} := \mathit{last} + 1; \\
\quad \mathit{used} := \mathit{used} \cup \{t\} \\
\textbf{procedure}\ \mathit{Release}\ (\textbf{value}\ t : \mathbb{N})\ \widehat{=} \\
\quad \mathit{used} := \mathit{used} \setminus \{t\}
\end{array}
$$

(e) Finally, give the result of diminishing the module by removing the abstract variable *used* (do not forget the initialization!). What is the proof obligation for this diminution?

****