TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

**EXAMINATIONS — 2005**

MID-YEAR

**COMP 426**

**Formal Software Development**

**Time Allowed:** 3 Hours

**Instructions:** Candidates should attempt **all FIVE** questions.
This exam will be marked out of 100.
Foreign language translation dictionaries are allowed.

## Question 1. Program verification [20 marks]

Consider the following program:

$$\textbf{if } y \leq m \textbf{ then } m := y \textbf{ else skip fi};$$
$$\textbf{if } z \leq m \textbf{ then } m := z \textbf{ else skip fi}$$

and the following pre- and postconditions:

Pre: $m = x$
Post: $m = min(x, y, z)$

Explain briefly how you would show that this program is correct with respect to the given pre- and postconditions:

**(a)** [10 marks] Using Floyd's inductive assertions method.

**(b)** [10 marks] Using Hoare logic.

In answering this question, you should emphasise the main differences between the two approaches and show how each method is applied to this particular problem. You should give the proof obligations generated using each approach, and explain how they are obtained. You do not need to give proofs for these proof obligations.

## Question 2. Weakest precondition semantics [20 marks]

**(a)** [6 marks] Define the following properties of a statement $S$, in terms of its weakest precondition semantics:

(i) monotonic (with respect to implication)

(ii) feasible (or strict)

(iii) terminating

(iv) disjunctive

(v) conjunctive

(vi) continuous

**(b)** [2 marks] Which of the above properties does Dijkstra require as *healthiness conditions*?

**(c)** [4 marks] Which of Dijkstra's healthiness conditions are **not** required in the refinement calculus? In each such case, explain why that property is not appropriate in a wide-spectrum language, and give an example of a construct in the refinement calculus which does not have that property.

**(d)** [8 marks] Dijkstra defines $wp$ for a do statement $DO = \mathbf{do}\ G \rightarrow S\ \mathbf{od}$ as:

$$wp(DO)(R) = \exists\, k \in \mathbb{N} \bullet H_k(R)$$

where $H_k$ is defined inductively as:

$$H_0(R) = \neg\, G \wedge R$$
$$H_{k+1}(R) = G \wedge wp(S)(H_k(R)) \vee H_0(R)$$

Show that do statements preserve feasibility (i.e. show that if $S$ is feasible, then so is $DO$).

## Question 3. Hoare logic and refinement [20 marks]

Some presentations of Hoare logic include the follow rule:

$$\frac{P_1 \{ S \} R_1 \quad P_2 \{ S \} R_2}{P_1 \wedge P_2 \{ S \} R_1 \wedge R_2}$$

**(a)** [8 marks] Prove that this law is sound (i.e. that the bottom correctness assertion below the line holds whenever the correctness assertions below the line both hold) for the class of programs that Hoare considers, using weakest preconditions.

For the purposes of this proof, you should assume that $S$ always terminates, so you can assume that $P_1 \{ S \} R_1$ holds if and only if $P \Rightarrow wp(S)(R)$ holds in all states.

Explain clearly each step in your proof and any properties of $wp(S)$ that you use.

**(b)** [4 marks] Discuss briefly how this law would be helpful in simplifying the verification of a sorting algorithm.

**(c)** [8 marks] Translating this law into the refinement calculus gives:

If $w\colon \left[ P_1 \Big/ R_1 \right] \sqsubseteq S$ and $w\colon \left[ P_2 \Big/ R_2 \right] \sqsubseteq S$, then $w\colon \left[ P_1 \wedge P_2 \Big/ R_1 \wedge R_2 \right] \sqsubseteq S$.

Is this a valid law in the refinement calculus? If so, prove that the law is valid. If not, explain why not and give a counter-example.

## Question 4. Refinement calculus [20 marks]

The following is an incomplete derivation of a program in the refinement calculus, where $d = f(n) \equiv (\exists\, i \in 2..n \bullet A[i] - A[i-1] = d) \wedge (\forall\, i \in 2..n \bullet A[i] - A[i-1] \leq d)$.

$$d\colon\left[ n \geq 2 \;\middle/\; d = f(n) \right] \tag{1}$$

(1) $\sqsubseteq$ ⟨ *introduce variable* ⟩

$$\begin{array}{l} [\![ \ \textbf{var}\ k \bullet \\ \quad d,k\colon\left[ n \geq 2 \;\middle/\; d = f(n) \right] \\ ]\!] \end{array} \tag{2}$$

(2) $\sqsubseteq$ ⟨ *split specification* ⟩

$$d,k\colon\left[ n \geq 2 \;\middle/\; Inv \right]\,; \tag{3}$$

$$d,k\colon\left[ Inv \;\middle/\; d = f(n) \right] \tag{4}$$

where $Inv \mathrel{\widehat{=}} 1 < k \leq n \wedge d = f(k)$

(3) $\sqsubseteq$ ⟨ *introduce assignment* ⟩

$d,k := A[2]{-}A[1], 2$

(4) $\sqsubseteq$ ⟨ *introduce* **do** ⟩

**do** $k \neq n \rightarrow$

$$d,k\colon\left[ Inv \wedge k \neq n \;\middle/\; Inv \wedge 0 \leq n{-}k{-}1 < n{-}k_0{-}1 \right] \tag{5}$$

**od**

(5) $\sqsubseteq$ ⟨ *following assignment* ⟩

$$d,k\colon\left[ Inv \wedge k \neq n \;\middle/\; Inv[k+1\,/\,k] \wedge 0 \leq n{-}(k{+}1){-}1 < n{-}k_0{-}1 \right]; \tag{6}$$

$k := k+1$

(6) $\sqsubseteq$ ⟨ *contract frame* ⟩

$$d\colon\left[ Inv \wedge k \neq n \;\middle/\; Inv[k+1\,/\,k] \right] \tag{7}$$

(Question 4 continued on next page)

**(Question 4 continued)**

**(a)** [3 marks] Explain what the specification statement labelled (1) means, identifying each of the components of the specification statement and explaining how it contributes to the meaning of the whole statement.

**(b)** [6 marks] For each step in the above derivation, give the *proof obligation(s)* that must be satisfied, and explain briefly why they are valid.

**(c)** [3 marks] What is the *loop invariant* used in the above derivation? Explain the rôle of the loop invariant in ensuring the correctness of the program.

**(d)** [3 marks] What is the *variant* or *bound* function used in the above derivation? Explain how it is used to ensure that the loop terminates.

**(e)** [5 marks] Show how the specification statements labelled (7) can be refined to code.

Indicate the laws used, write down the necessary proof obligations and briefly explain why they are valid.

## Question 5. Data refinement [20 marks]

Consider the following module, *Alloc*, which specifies a resource allocation mechanism. Resources are represented by natural numbers, and the total number of resources available is given by the global constant $N$. A call on *Get* returns a set of $n$ resources, provided that sufficient resources are available; otherwise the empty set is returned. A call on *Free* returns a set of resources to the allocator so that they can be reused.

(Note that $\mathbb{P}$ denotes power set, $\#s$ gives the size of the (finite) set $s$, and $\setminus$ denotes set subtraction.)

$$
\begin{array}{l}
\textbf{module}\ \mathit{Alloc} \mathrel{\widehat{=}} \\
\quad \textbf{var}\ \mathit{free} : \textbf{set of}\ \mathbb{N} := 1 \mathinner{..} N\ \textbf{;} \\
\quad \textbf{procedure}\ \mathit{Get}(\textbf{value}\ n : \mathbb{N}, \textbf{result}\ r : \mathbb{P}\,\mathbb{N}) \mathrel{\widehat{=}} \\
\qquad \textbf{if}\ \ n \leq \#\mathit{free} \rightarrow \\
\qquad\quad r, \mathit{free} \mathbin{\boldsymbol{:}} \left[\#\mathit{free} \leq n \;\middle/\; r \subseteq \mathit{free}_0 \wedge \#r = n \wedge \mathit{free} = \mathit{free}_0 \setminus r\right]\ \textbf{;} \\
\qquad [\!]\ n > \#\mathit{free} \rightarrow \\
\qquad\quad r := \varnothing \\
\qquad \textbf{fi ;} \\
\quad \textbf{procedure}\ \mathit{Free}(\textbf{value}\ r : \mathbb{P}\,\mathbb{N}) \mathrel{\widehat{=}} \\
\qquad \mathit{free} := \mathit{free} \cup r\ \textbf{;} \\
\textbf{end}
\end{array}
$$

Also consider the following module, *Alloc*1, which implements the above resource allocator. Variable $m$ is always greater than all numbers that have been allocated, and $a$ represents the number of resources currently available (the total number of resources that have ever been allocated minus the total number of resources that have been freed). A call on *Get*1 returns the contiguous set of naturals from $m$ to $m{+}n{-}1$ (inclusive), increases $m$ by $n$ and decreases $a$ by $n$. A call on *Free*1 just increases $a$ by $n$.

$$
\begin{array}{l}
\textbf{module}\ \mathit{Alloc}1 \mathrel{\widehat{=}} \\
\quad \textbf{var}\ m : \mathbb{N} := 1, a : \mathbb{N} := N\ \textbf{;} \\
\quad \textbf{procedure}\ \mathit{Get}(\textbf{value}\ n : \mathbb{N}, \textbf{result}\ r : \mathbb{P}\,\mathbb{N}) \mathrel{\widehat{=}} \\
\qquad \textbf{if}\ \ n \leq a \rightarrow \\
\qquad\quad r,\ m,\ a := n \mathinner{..} n{+}m{-}1,\ m{+}n,\ a{-}n\ \textbf{;} \\
\qquad [\!]\ \#\mathit{free} \leq n \rightarrow \\
\qquad\quad r := \varnothing \\
\qquad \textbf{fi ;} \\
\quad \textbf{procedure}\ \mathit{Free}(\textbf{value}\ r : \mathbb{P}\,\mathbb{N}) \mathrel{\widehat{=}} \\
\qquad a := a + \#r\ \textbf{;} \\
\textbf{end}
\end{array}
$$

(Question 5 continued on next page)

**(Question 5 continued)**

**(a)** [10 marks] Explain how the technique of *data refinement* can be used to show that *Alloc*1 correctly implements *Alloc*. Give an *abstraction invariant* (or *retrieve relation*) relating the two modules, and give the necessary proof obligations. Give a brief argument to show that the proof obligations hold — you do not need to give detailed proofs.

**(b)** [6 marks] Explain how a concrete module can be calculated from an abstract one, and give the concrete module that would be obtained by applying this approach to the *Alloc* module using your abstraction invariant from part **(a)**.

What is the relationship between the calculated module and the module *Allocator*1 given above?

**(c)** [4 marks] Some authors claim that the abstraction relation used in data refinement should be a surjective (onto) function from the concrete state to the abstract state — this is sometimes encapsulated in the slogan "no confusion and no junk". Outline the main arguments for and against this claim.

**Note: You may answer this question either in terms of the refinement calculus, or in terms of Z.**

*******************************