PRIFYSGOL CYMRU; UNIVERSITY OF WALES

DEGREE EXAMINATIONS MAY/JUNE 2003

SWANSEA

Computer Science

CS 213 System Specification

Attempt 2 questions out of 3

Time allowed: 2 hours

Students are permitted to use the dictionaries provided by the University

Students are NOT permitted to use calculators

CS_213
SYSTEM SPECIFICATION

(Attempt 2 questions out of 3)

Question 1

(a) The following is a simple Maude module.

```
1. fmod BASIC-NAT is
2.     sort Nat .

3.     op 0 : -> Nat .
4.     op s : Nat -> Nat .
5.     op _+_ : Nat Nat -> Nat .

6.     vars N M : Nat .

7.     eq 0 + N = N .
8.     eq s(M) + N = s(M + N) .
9. endfm
```

Briefly, explain what each numbered line means. In the usual definition of the natural numbers, line 5 would normally look like this:

```
  op _+_ : Nat Nat -> Nat [property1 property2] .
```

What are *property1* and *property2*? **[10 Marks]**

(b) The following Maude module represents part of the definition of an *array* in Maude. However, the *defining equations* are missing. What are they? **[5 Marks]**

```
fmod ARRAY is

    sorts Array Index Data .

    op _[_] : Array Index -> Data .
    op _[_/_] : Array Data Index -> Array .

    var A : Array .
    vars I J : Index .
    var D : Data .

    *** Equations should go here
endfm
```

(c) The following two Maude code fragments define *eight-bit bytes*, under the assumption that sorts `Bits` representing *arbitrary-length bit-strings,* and `Bit` representing *a single bit,* already exists.

Fragment (a)

```
sort Byte .
subsort Byte < Bits .

vars b1 b2 b3 b4 b5 b6 b7 b8 : Bit .

mb (b1 b2 b3 b4 b5 b6 b7 b8) : Byte .
```

Fragment (b)

```
sort Byte .
subsort Byte < Bits .

var B : Bits .

cmb B : Byte if |B| = 8 .
```

The operator `|_| : Bits -> MachineInt` returns the *length* of a bit-string. Both fragments 'work', but one is a much better solution than the other. Which one, and why? **[5 Marks]**

Question Continues Overleaf.

(d) The following Maude module represents a stack.

```
fmod STACK is

    sorts Stack Elt .

    op EmptyStack : -> Stack .
    op ErrorElt : -> Elt .

    op push : Stack Elt -> Stack .
    op pop : Stack -> Stack .
    op top : Stack -> Elt .
    op isEmpty : Stack -> Bool .

    var S : Stack .
    var E : Elt .

    eq isEmpty(EmptyStack) = true .
    eq isEmpty(push(S,E)) = false .

    eq top(EmptyStack) = ErrorElt .
    eq top(push(S,E)) = E .

    eq pop(EmptyStack) = EmptyStack .
    eq pop(push(S,E)) = S .
endfm
```

Add a new operator *swap* to the module that swaps the top two elements on the stack. That is, suppose that the top element of stack *S* is *A,* and the element below that is *B*. After the application of *swap* to *S,* the top element is *B*, and the element below that is *A*. The remainder of the stack is unaltered. You will need to consider what happens when you try to apply swap to a stack that only contains one element. **[5 Marks]**

Question 2

Consider a simple microprocessor with 16 registers each of 16-bits, a memory of 64K 16-bit words (i.e. memory addresses are 16-bits long), and a 16-bit program counter. Instructions are 16-bits long: the first four bits are the *opcode;* the second four bits code *register A;* the third four bits code *register B;* and the final four bits code *register C*. There are four instructions

- `Add Ra, Rb, Rc` – Add registers A and B, storing the result in register C. Increment the program counter by 1. The *opcode* is `0000`.
- `Sub Ra, Rb, Rc` – Subtract register A from register B, storing the result in register C. Increment the program counter by 1. The *opcode* is `0001`.
- `Load Ra, Rb, Rc` – Add registers A and B. Use the result as a memory address and read a word from memory, storing the result in register C. Increment the program counter by 1. The *opcode* is `0010`.
- `Store Ra, Rb, Rc` – Add registers A and B. Use the result as a memory address and write register C to memory. Increment the program counter by 1. The *opcode* is `0011`.
- All other *opcodes* are *invalid,* and result in the program counter being set to: 1111111111111111 – that is, the top-most word in memory.

You may assume the following are available.

- The module `BINARY`, containing sorts `Bits` (arbitrary-length bit-strings) and `Bit` (single bits), and the usual range of logic and arithmetic operations.
- Modules `MEM` and `REG` defining sorts `Mem` and `Reg` representing memory and registers, as well as read and write operations on memory and registers.
- Sorts `HalfWord` of 16-bit words, and `Nibble` of 4-bit words have been defined in terms of `Bits` and `Bit`.
- The operations `opcode rega regb regc: HalfWord -> Nibble` that extract the opcode, register A, register B and register C fields from an instruction.
- Addition and subtraction operations on 16-bit words `_+_ _-_ : HalfWord HalfWord -> HalfWord` .
- The built-in module `MACHINE-INT`.

(a) Write a Maude module defining the *state* of the microprocessor. Your module should contain appropriate *tupling* and *projection* operators. Minor syntactic errors will not be penalised. **[5 Marks]**

(b) Write a Maude module defining the *behaviour* of the microprocessor as an *iterated map*. Your module should contain the *iterated map function,* as well as the *next-state* function. Minor syntactic errors will not be penalised. **[10 Marks]**

Question Continues Overleaf.

(c) The current instruction set of the microprocessor is inadequate because there are no *flow control* instructions. Informally define four instructions that do the following.

- Unconditional Jump.
- Conditional Jump.
- Subroutine Call.
- Return from Subroutine.

It is suggested that you retain the same format (i.e. *opcode, registerA, registerB, registerC*) of the existing instructions. However, you may find that in some cases not all the fields are necessary.

Extend your answer to part (b) to incorporate Maude representations of the new instructions. Minor syntactic errors will not be penalised. **[10 Marks]**

Question 3

(a) Explain informally, perhaps with the help of some simple examples, what it means for a term rewriting system to be *terminating* and *confluent*. **[5 Marks]**

(b) Consider the following simple Maude module.

```
fmod BASIC-NAT is
    sort Nat .

    op 0 : -> Nat .
    op s : Nat -> Nat .
    op _+_ : Nat Nat -> Nat .

    vars N M : Nat .

    eq 0 + N = N .
    eq s(M) + N = s(M + N) .
endfm
```

Explain how the following expression would be re-written in Maude, using the module above:

```
s(s(s(0))) + s(s(0))
```

Consider the following expression, in which N is a variable of sort Nat:

```
(s(s(0)) + N) == (N + s(s(0)))
```

What does this reduce to and why?

[5 Marks]

(c) The following Maude module represents a *multiplexor.*

```
fmod MUX is
    protecting MACHINE-INT .

    sort BoolStr .

    op _(_) : BoolStr MachineInt -> Bool .
    op mux : BoolStr BoolStr BoolStr -> BoolStr .

    vars X Y B : BoolStr .
    var T : MachineInt .

    ceq mux(X,Y,B)(T) = X(T) if B(T) == true .
    ceq mux(X,Y,B)(T) = Y(T) if B(T) == false .
endfm
```

Modify the multiplexor module so that outputs at time T are the result of inputs that arrive at time T—1 instead of time T. **[5 Marks]**

(d) The following is an informal description of a simple *counter.* The output of the counter is a *stream of Integers.* The inputs to the counter are two *streams of Booleans.*

- stopStart – when a *true* element arrives on this stream at time T, the counter *stops.* That is, the output at time T is equal to the output at time T-1. Otherwise, the output at time T will be either one greater or one less than the output at time T-1 (depending on the value on the stream upDown below).
- upDown – when a true element arrives on this stream at time T, the output of the counter at time T will be one greater than the output at time T-1, provided the counter is not *stopped.* Otherwise, the output at time T will be one less than the output at time T-1, provided the counter is not *stopped.*
- At time T == 0, the output of the counter is 0 – *regardless of the values on the input streams.*

Write a Maude module formally defining the behaviour of the counter. Minor syntactic errors will not be penalised. **[10 Marks]**