TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI

VICTORIA
UNIVERSITY OF WELLINGTON

EXAMINATIONS – 2013

TRIMESTER 2

SWEN 430

Compiler Engineering

Time Allowed: THREE HOURS

Instructions:

- Closed Book.
- This examination will be marked out of **180** marks.
- Answer all questions.
- You may answer the questions in any order. Make sure you clearly identify the question you are answering.
- No calculators are permitted.
- Non-electronic Foreign language dictionaries are allowed.

Question	Topic	Marks
1.	Parsing & Semantics	30
2.	Typing	30
3.	Java Bytecode	30
4.	Machine Code	30
5.	Dataflow Analysis	30
6.	Advanced Topics	30
	Total	180

Question 1. Parsing & Semantics [30 marks]

Consider the following variant of the *lambda calculus* which has been extended with **`if`** statements, but is incomplete with respect to the evaluation rules.

$$
\begin{array}{lll}
\mathtt{t} ::= & & \text{(Terms)} \\
\quad | & \mathtt{x} & \text{(Variables)} \\
\quad | & \mathtt{v} & \text{(Values)} \\
\quad | & \mathtt{t_1\ t_2} & \text{(App)} \\
\quad | & \mathtt{if(t_1)\ \{t_2\}\ else\ \{t_3\}} & \text{(App)} \\
 & & \\
\mathtt{v} ::= & & \text{(Values)} \\
\quad | & \lambda\mathtt{x.t} & \text{(Function)} \\
\quad | & \mathtt{true} & \text{True} \\
\quad | & \mathtt{false} & \text{False}
\end{array}
$$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1'}}{\mathtt{t_1\ t_2} \longrightarrow \mathtt{t_1'\ t_2}} \quad \text{(R-App1)}$$

$$\frac{\mathtt{t_2} \longrightarrow \mathtt{t_2'}}{\mathtt{v_1\ t_2} \longrightarrow \mathtt{v_1\ t_2'}} \quad \text{(R-App2)}$$

$$\frac{}{(\lambda\mathtt{x.t_1})\ \mathtt{v_1} \longrightarrow \mathtt{t_1}[\mathtt{x} \mapsto \mathtt{v_1}]} \quad \text{(R-App3)}$$

$$\frac{}{\mathtt{if(true)\ \{t_1\}\ else\ \{t_2\}} \longrightarrow \mathtt{t_1}} \quad \text{(R-If1)}$$

$$\frac{}{\mathtt{if(false)\ \{t_1\}\ else\ \{t_2\}} \longrightarrow \mathtt{t_2}} \quad \text{(R-If2)}$$

(a) Using the above rules, evaluate each of the following terms as much as possible.

(i) [2 marks] $\mathtt{if(true)\ \{true\}\ else\ \{false\}}$

(ii) [2 marks] $(\lambda\mathtt{x.x})\ \mathtt{true}$

(iii) [2 marks] $(\lambda\mathtt{y.y})\ \mathtt{x}$

(iv) [2 marks] $(\lambda\mathtt{x.if(x)\ \{y\}\ else\ \{z\}})\ \mathtt{true}$

(v) [2 marks] $\lambda\mathtt{y.}\Big((\lambda\mathtt{x.if(x)\ \{y\}\ else\ \{z\}})\ \mathtt{true}\Big)$

(b) [5 marks] The evaluation rules given above are incomplete for `if` statements. For example, the following term does not evaluate to `true` as expected:

$$\mathtt{if\big(if(true)\ \{true\}\ else\ \{false\}\big)\ \{true\}\ else\ \{false\}}$$

Give an appropriate evaluation rule for terms of the form $\mathtt{if(t_1)\ \{t_2\}\ else\ \{t_3\}}$ when the condition $\mathtt{t_1}$ is neither the literal `true` nor `false`.

(c) [10 marks] Sketch suitable Java classes for representing the *Abstract Syntax Tree (AST)* of a program in the above lambda calculus variant. In your sketch you should include fields, constructors and inheritance relationships, but need not show anything else.

(d) [5 marks] A well-known issue with the lambda-calculus is the *variable capture* problem. Briefly, discuss what this is using an example to illustrate.

Question 2. Typing [30 marks]

Consider the following description of the *simply-typed lambda calculus*:

$$
\begin{array}{lll}
\mathtt{t} ::= & & \text{(Terms)} \\
\quad \mid \mathtt{x} & & \text{(Variables)} \\
\quad \mid \mathtt{v} & & \text{(Values)} \\
\quad \mid \mathtt{t\ t} & & \text{(Apps)} \\
\\
\mathtt{v} ::= & & \text{(Values)} \\
\quad \mid \lambda \mathtt{x} : \mathtt{T}.\mathtt{t} & & \text{(Function)} \\
\quad \mid \mathtt{c} & & \text{(Integer)} \\
\\
\mathtt{T} ::= & & \text{(Types)} \\
\quad \mid \mathtt{T} \rightarrow \mathtt{T} & & \text{(Fun type)} \\
\quad \mid \mathtt{int} & & \text{(Int type)}
\end{array}
$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \quad \text{(R-App1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \quad \text{(R-App2)}$$

$$\frac{}{(\lambda \mathtt{x} : \mathtt{T}.\mathtt{t}_1)\ \mathtt{v}_1 \longrightarrow \mathtt{t}_1[\mathtt{x} \mapsto \mathtt{v}_1]} \quad \text{(R-App3)}$$

(a) Consider the following rule for typing function applications:

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \rightarrow \mathtt{T}_2 \quad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_1}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_2} \quad \text{(T-App)}$$

(i) [2 marks] The *typing environment* is represented by Γ. Briefly, describe what this is.

(ii) [5 marks] In your own words, describe the above typing rule for function applications (T-App).

(iii) [3 marks] Give an appropriate typing rule for *variables*.

(iv) [5 marks] Give an appropriate typing rule for *lambdas* (i.e. terms of the form $\lambda \mathtt{x} : \mathtt{T}_1.\mathtt{t}_2$).

(b) A term in the lambda calculus is said to be *stuck* if it is not yet a value and cannot be further evaluated.

(i) [3 marks] Based on the rules above, give an example of a lambda term which is stuck.

(ii) [3 marks] Briefly, discuss whether or not stuck terms can be typed in the simply-typed lambda calculus.

(iii) [4 marks] In your own words, briefly discuss the purpose of a *type system*. You may refer to rules and examples given in this question.

(iv) [5 marks] The so-called *progress* and *preservation* theorems establish important properties which are often required of a type system. Briefly, state what these properties are.

Question 3. Java Bytecode [30 marks]

(a) Consider the following method written in Java bytecode:

```
Number method(Integer, Double);
   0:   aload_1
   1:   ifnonnull 9
   4:   aload_2
   5:   astore_3
   6:   goto    11
   9:   aload_1
   10:  astore_3
   11:  aload_3
   12:  areturn
```

(i) [3 marks] What is the *maximum stack height* of this method?

(ii) [5 marks] Give Java source code which is equivalent to the above method.

(iii) [5 marks] Rewrite the above method to an equivalent which uses as few bytecodes as possible.

(iv) [3 marks] Explain the difference between *absolute addressing* and *relative addressing*.

(b) Each of the following JVM error messages is reported when a particular error is encountered in a JVM **`class`** file. For each, briefly discuss what kind of error might have caused the problem. You may use examples to illustrate as necessary.

(i) [2 marks] "`Expecting to find float on stack, found int`"

(ii) [2 marks] "`Unable to pop operand off an empty stack`"

(iii) [2 marks] "`Accessing value from uninitialized register`"

(iv) [2 marks] "`Illegal target of jump or branch`"

(v) [2 marks] "`Stack overflow`"

(vi) [2 marks] "`Inconsistent stack height`"

(vii) [2 marks] "`Falling off the end of the code`"

Question 4. Machine Code [30 marks]

(a) [5 marks] When generating machine code, *calling conventions* are needed to coordinate between *caller* and *callee*. Briefly, discuss why a calling convention is necessary.

(b) [5 marks] Briefly, discuss how *stack frames* are organised. For simplicity, assume that parameters and return values are always passed on the stack.

(c) [10 marks] Translate the following method into x86_64 machine code. You should assume:

- Parameters `x` and `y` are passed in registers `rdi` and `rsi` respectively.
- The return value is passed in register `rax`.
- All other registers are callee-saved.

NOTE: the Appendix on page 9 provides an overview of x86_64 machine instructions for reference.

```
1  int foo(int x, int y){
2    if (y < 9){
3      x = 8;
4    }
5    int z = x + y;
6    return z;
7  }
```

(d) *Jump tables* are used to implement **`switch`** statements efficiently.

(i) [5 marks] Briefly, discuss what a jump table is.

(ii) [5 marks] Briefly, discuss why jump tables can be more efficient than sequences of conditional branches.

Question 5. Dataflow Analysis [30 marks]

(a) An integral component of any register allocation mechanism is *live variables analysis*.

(i) [5 marks] Briefly, discuss what live variable analysis is.

(ii) [5 marks] Draw the *interference graph* for the following method:

```
int a = 9;
int b = 8;
int c = a + b;
b = c + a;
a = c + b;
c = 8;
b = 9 * c;
```

(iii) [2 marks] Give a *minimal colouring* for your interference graph.

(iv) [5 marks] Briefly, discuss how graph colouring helps with register allocation.

(b) [5 marks] Briefly, discuss the term *definite assignment*. You may use examples to illustrates as necessary.

(c) A common phase used in a compiler is *deadcode elimination*.

(i) [3 marks] Give a short example illustrating some *dead code*.

(ii) [5 marks] Briefly, describe an algorithm for identifying dead code.

Question 6. Advanced Topics [30 marks]

(a) Class Hierarchy Analysis is a technique for approximating a program's call graph.

(i) [5 marks] Briefly, discuss how Class Hierarchy Analysis works.

(ii) [5 marks] *Devirtualisation* is a technique where dynamically-dispatched messages are replaced with direct procedure calls. Briefly, discuss why class hierarchy analysis is useful for this.

(b) Garbage collection is a technique used for memory management, for example, in the Java Virtual Machine.

(i) [5 marks] Briefly, outline one approach to implementing a garbage collector.

(ii) [5 marks] Briefly, outline one common challenge faced by a garbage collector.

(c) Copy elimination is a technique for improving performance in languages with *value semantics* (such as the WHILE language discussed in lectures).

(i) [5 marks] Briefly, discuss why copy elimination could improve performance in the WHILE language.

(ii) [5 marks] Briefly, discuss one approach to copy elimination that could be used for the WHILE language.

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

This page is intentionally left blank so that you may remove the appendix page.

Appendix

`movq $c, %rax`	Assign constant `c` to `rax` register
`movq %rax, %rdi`	Assign register `rax` to `rdi` register
`addq $c, %rax`	Add constant `c` to `rax` register
`addq %rax, %rbx`	Add `rax` register to `rbx` register
`subq $c, %rax`	Substract constant `c` from `rax` register
`subq %rax, %rbx`	Subtract `rax` register from `rbx` register
`cmpq $0, %rdx`	Compare constant `0` register against `rdx` register
`cmpq %rax, %rdx`	Compare `rax` register against `rdx` register
`movq %rax, (%rbx)`	Assign `rax` register to dword at address `rbx`
`movq (%rbx),%rax`	Assign `rax` register from dword at address `rbx`
`movq 4(%rsp),%rax`	Assign `rax` register from dword at address `rsp+4`
`movq %rdx, (%rsi,%rbx,4)`	Assign `rdx` register to dword at address `rsi+4*rbx`
`pushq %rax`	Push `rax` register onto stack
`pushq %c`	Push constant `c` onto stack
`popq %rdi`	Pop qword off stack and assign to register `rdi`
`jz target`	Branch to `target` if zero flag set.
`jnz target`	Branch to `target` if zero flag not set.
`jl target`	Branch to `target` if less than (i.e. sign flag set).
`jle target`	Branch to `target` if less than or equal (i.e. sign or zero flags set).
`ret`	Return from function.