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

EXAMINATIONS — 2014

TRIMESTER 1

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.	WHILE Language	30
2.	Definite Unassignment	30
3.	Java Bytecode	30
4.	Machine Code	30
5.	Static Single Assignment	30
6.	Interprocedural Analysis	30
	Total	180

Question 1. WHILE Language [30 marks]

The λ_W is a simplified representation of a WHILE programming language as used in the lectures on semantics:

p	::=	$\texttt{f}_1 \ldots \texttt{f}_\texttt{n}\ \texttt{e}$	*programs*
f	::=	$\texttt{T}_1\ \texttt{n}_1(\texttt{T}_2\ \texttt{n}_2)\ \{\ \overline{\texttt{s}}\ \}$	*functions*
e	::=		*expressions*
	|	v	*constants*
	|	n	*variables*
	|	$\texttt{e}_1\ \texttt{op}\ \texttt{e}_2$	*binary*
	|	$\texttt{e}_1(\texttt{e}_2)$	*application*
	|	$\overline{\texttt{s}}$	*function body*
s	::=	$\texttt{n} = \texttt{e};$ | return e;	*statements*
v	::=	null | true | false | λn.$\overline{\texttt{s}}$	*values*
		| ... | -1 | 0 | 1 | ...	
T	::=	bool | int | null | $\texttt{T}_1 \vee \texttt{T}_2$	*types*
op	::=	$'<' \mid '<=' \mid '>=' \mid '>'$	*operators*
		$\mid '!=' \mid '==' \mid '+' \mid '-' \mid '*'$	

(a) For each of the following little programs state if it is a syntactically correct program in the λ_W language and also state if it is also type safe. Justify your answer:

(i) [2 marks] `42 + 33`

(ii) [2 marks] `42 + 33 + 56`

(iii) [2 marks] `int f(bool x) { x = true; return f(true); } 0`

(iv) [3 marks] `bool f(int x) { return x == false; } 0`

(v) [3 marks] `bool f(int x) { return x = x = x = return 42;;; } 0`

(vi) [3 marks] `int f(int x) { x * x; return 3; } 0`

(b) [5 marks] What is meant by the *operational semantics* and why would we want one for a language?

(c) [10 marks] Describe the subtyping relation used by the WHILE language by stating the subtyping rules for the types listed above as well as records: $\{\overline{\texttt{T n}}\}$. Don't forget about union types, here are two basic reflexivity and transitivity rules to remind you of the format to use for writing down the rules stating that "one type is a subtype of another type":

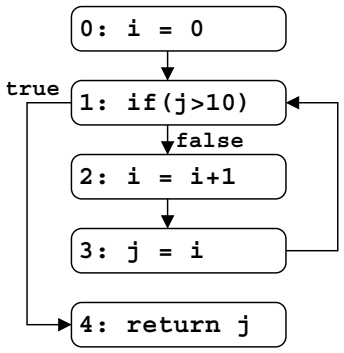
$$\frac{}{T_1 \leq T_1}$$

$$\frac{T_1 \leq T_2 \quad T_2 \leq T_3}{T_1 \leq T_3}$$

Question 2. Definite Unassignment [30 marks]

The *definite unassignment* phase is used in Java to check that **`final`** variables are only assigned once. The following illustrates using a Java bytecode-like language:

```
int aMethod() {
   var i;
   final var j;
   i = 0;
 lab1:
   if j > 10 goto lab2;
   i = i+1;
   j = i;
   goto lab1;
 lab2:
   return j;
}
```



The above method fails definite unassignment because the **`final`** variable `j` *may be assigned more than once*. The definite unassignment algorithm determines, at each point, which variables *may have been assigned* at that point. In particular, the right part of the figure uses the Control Flow Graph (CFG) representation of the program on the left.

Going back to the WHILE language, imagine that we extended its syntax to include **`final`** variables.

(a) [10 marks] Please described how you would implement the *definite unassignment* algorithm following the approach you took in assignment 2 (i.e. **without using a CFG**). Include a pseudocode for the core of the algorithm and basic descriptions of the data structures used.

(b) [10 marks] Now, please describe how would you design and implement a simple dataflow analysis for *definite unassignment* **utilising a CFG**. Include a core algorithm in pseudocode and a brief description of the CFG representation that you would use (just names of classes and relationships representing different parts of a CFG would suffice - around one paragraph of writing for the description of CFG representation at most). Discuss whether your algorithm requires multiple iterations - why or why not.

(c) [10 marks] Discuss and justify why do you think your algorithm that uses a CFG will terminate and work correctly? Please refer to our class discussion about a lattice that the algorithm operates on.

Question 3. Java Bytecode [30 marks]

(a) Consider the following Java bytecode, which fails *bytecode verification*:

```
boolean method(int);
   0: iconst_2
   1: istore_2
   2: iload_2
   3: aload_1
   4: if_icmpeq 13
   7: iconst_0
   8: dup
   9: goto 14
  13: iconst_1
  14: pop
  15: pop
  16: ireturn
```

(i) [5 marks] Briefly discuss the different ways in which this method fails bytecode verification.

(ii) [5 marks] The bytecode verifier employs a *least upper bound* operation on types. Discuss what this means, and how it relates to bytecode verification. Use examples to illustrate as necessary.

(iii) [5 marks] A bytecode method can fail verification because the verifier claims it uses types incorrectly, when at runtime it does not. Using an example to illustrate, discuss how this can occur.

(b) [5 marks] Consider the following bytecode, which creates and initialises a new object.

```
0:   new java/lang/Integer
3:   dup
4:   iload_0
5:   invokespecial java/lang/Integer."<init>":(I)V
8:   ...
```

An object can only be used *once it is initialised* (i.e. the `<init>` method has been called upon it). Briefly discuss why this complicates bytecode verification.

(c) [5 marks] Describe the difference between `tableswitch` and `lookupswitch` bytecodes and specify which is going to be more appropriate for different kinds of possible `switch` statements.

(d) [5 marks] Answer one of the following questions (it is up to you to decide which one!):

- What is the difference between `invokevirtual` and `invokedynamic`?
- Describe, by using an example, the difference between *single* dispatch and *multiple* dispatch.

Question 4. Machine Code [30 marks]

Consider the following function written in `x86_64` assembly language:

```
1  foo:
2      pushq   %rbp
3      movq    %rsp, %rbp
4      movl    %edi, -20(%rbp)
5      movl    %esi, -24(%rbp)
6      movl    -20(%rbp), %eax
7      cmpl    -24(%rbp), %eax
8      jge .L2
9      movl    -24(%rbp), %eax
10     movl    %eax, -4(%rbp)
11     jmp .L3
12 .L2:
13     movl    -20(%rbp), %eax
14     movl    %eax, -4(%rbp)
15 .L3:
16     movl    -4(%rbp), %eax
17     popq    %rbp
18     ret
```

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

(a) [5 marks] Function parameters are normally passed *on the stack* or *in registers*. How are parameters passed in the above function? Justify your answer.

(b) [5 marks] Draw the layout of the *stack frame* for this function using relative offsets (in bytes) for each component.

(c) [5 marks] The special *flags register* is an unusual feature of the `x86_64` architecture. Briefly, explain what this register does using examples from above to illustrate.

(d) [5 marks] In your own words, explain what this function does.

(e) [5 marks] Outline how the above program can be rewritten to reduce the number of machine instructions required.

(f) [5 marks] The above program is not particularly *efficient*. Briefly, discuss why this is.

Question 5. Static Single Assignment [30 marks]

(a) [5 marks] Briefly, discuss what *static single assignment* form is.

(b) [5 marks] Briefly, discuss what φ-nodes are used for in SSA form.

(c) Consider the following WHILE function:

```
int f(int x, int y) {
  int z;

  if x < y {
    z = x;
  } else {
    z = y;
  }

  return z + x + y;
}
```

(i) [5 marks] Draw the *control-flow graph* of this function.

(ii) [5 marks] Translate this function into static single assignment form.

(d) Algorithms for computing static single assignment form consider the *dominance structure* of nodes in the control-flow graph.

(i) [5 marks] Briefly, explain what it means for one node to dominate another in a control-flow graph

(ii) [5 marks] Briefly, explain how the dominance structure of nodes in a control-flow graph helps in computing static single assignment.

Question 6. Interprocedural Analysis [30 marks]

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

(i) [10 marks] Using examples where appropriate, discuss how Class Hierarchy Analysis works.

(ii) [5 marks] Briefly discuss how Class Hierarchy Analysis can be used to improve program performance.

(iii) [5 marks] In Java, it is impossible to know exactly which classes can be used by a given program. This is because classes can be loaded at runtime using the `ClassLoader`. Discuss how this affects the validity of Class Hierarchy Analysis.

(b) *Variable Type Analysis* is another technique for approximating a program's call graph.

(i) [5 marks] Variable Type Analysis uses a *type propagation graph*. Using examples where appropriate, discuss how this works.

(ii) [5 marks] Briefly discuss why Variable Type Analysis gives more precise results than Class Hierarchy Analysis.

****

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.