TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI

VUW VICTORIA UNIVERSITY OF WELLINGTON

EXAMINATIONS — 2007

END YEAR

COMP471

COMPILER TECHNOLOGIES

Time Allowed: 3 Hours

Instructions:

- *Read each question carefully before attempting it.*
- 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.
- Non-electronic foreign-language English dictionaries are permitted.
- Reference material is NOT PERMITTED.
- Calculators are NOT PERMITTED.
- Use of mobile phones, laptop computers, PDAs or other electronic devices is NOT PERMITTED.

Question 1. Preliminaries [25 marks]

(a) [10 marks] Why do we need to prove *subject reduction* and what does it show?

(b) [10 marks] Why do we need to prove *progress* and what does it show?

(c) [5 marks] In your own words, discuss the purpose of a *type system*.

Question 2. Untyped λ-Calculus [47 marks]

Consider the following language:

$$
\begin{array}{lcll}
\mathtt{t} & ::= & & \text{(Terms)} \\
 & | & \mathtt{x} & \text{(Variables)} \\
 & | & \mathtt{v} & \text{(Values)} \\
 & | & \mathtt{t\ t} & \text{(App)} \\
 & & & \\
\mathtt{v} & ::= & & \text{(Values)} \\
 & | & \lambda \mathtt{x.t} & \text{(Function)} \\
 & | & \mathtt{c} & \text{(Integer)}
\end{array}
$$

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

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

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

(a) [5 marks] State the subject reduction and progress theorems for this language.

(b) [15 marks] Prove the subject reduction theorem.

(c) [15 marks] Prove the progress theorem.

The *size* of a term $\mathtt{t}$, written $\mathit{size}(\mathtt{t})$, is defined as follows:

$$\mathit{size}(\mathtt{x}) = 1$$

$$\mathit{size}(\mathtt{c}) = 1$$

$$\mathit{size}(\lambda \mathtt{x.t}) = \mathit{size}(\mathtt{t}) + 1$$

$$\mathit{size}(\mathtt{t_1\ t_2}) = \mathit{size}(\mathtt{t_1}) + \mathit{size}(\mathtt{t_2})$$

The set of *free variables* of a term $\mathtt{t}$, written $FV(\mathtt{t})$, is defined as follows:

$$FV(\mathtt{x}) = \{\mathtt{x}\}$$

$$FV(\mathtt{c}) = \{\}$$

$$FV(\lambda \mathtt{x.t_1}) = FV(\mathtt{t_1})\ \{\mathtt{x}\}$$

$$FV(\mathtt{t_1\ t_2}) = FV(\mathtt{t_1}) \cup FV(\mathtt{t_2})$$

(d) [12 marks] Give a careful proof that $|FV(\mathtt{t})| \leq \mathit{size}(\mathtt{t})$ for every term $\mathtt{t}$.

Question 3. Type Rules [18 marks]

Consider the following Simply Typed λ-Calculus with the pair expression extension (here types include simple types ($\mathtt{T}$) and types of functions ($\mathtt{T} \to \mathtt{T}$); the variable types are contained the the environment Γ):

$\mathtt{t}$	::=		(Terms)
	|	$\mathtt{x}$	(Variables)
	|	$\mathtt{v}$	(Values)
	|	$\mathtt{t\ t}$	(App)
	|	$\{\mathtt{t}, \mathtt{t}\}$	(Pair)
	|	$\mathtt{t.1}$	(First)
	|	$\mathtt{t.2}$	(Second)
$\mathtt{v}$	::=		(Values)
	|	$\lambda \mathtt{x} : \mathtt{T}.\mathtt{t}$	(Function)
	|	$\mathtt{c}$	(Integer)
	|	$\{\mathtt{v}, \mathtt{v}\}$	(Pair value)

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

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

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

$$\frac{}{\vdash \mathtt{n} : \mathtt{int}} \quad \text{(T-Int)}$$

$$\frac{\mathtt{x} : \mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \quad \text{(T-Var)}$$

$$\frac{\Gamma \cup \{\mathtt{x} : \mathtt{T}_1\} \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x} : \mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \quad \text{(T-Fun)}$$

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

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \quad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \{\mathtt{t}_1, \mathtt{t}_2\} : \mathtt{T}_1 \times \mathtt{T}_2} \quad \text{(T-Pair)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \times \mathtt{T}_{12}}{\Gamma \vdash \mathtt{t}_1.1 : \mathtt{T}_{11}} \quad \text{(T-Proj1)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \times \mathtt{T}_{12}}{\Gamma \vdash \mathtt{t}_1.2 : \mathtt{T}_{12}} \quad \text{(T-Proj2)}$$

(a) [18 marks] The *typing rules* for all expressions are given above. However, some of the *evaluation rules* are missing. Please provide the **six** missing evaluation rules: two for each kind of projection (one is defining a reduction and one is defining the order) and two for specifying the order of evaluation inside a pair.

Bytecode Language

In the remaining questions, the following language (which is similar to Java Bytecode) will be used:

c	integers
v, m, f, C	local variable, method, field and class names (respectively)
e, e_1, e_2	arithmetic expressions
aop	binary arithmetic operators (e.g. `+`,`-`,`*`,`/`)
bop	binary boolean operators (e.g. `==`,`<`,`>`,`<=`,`>=`)
L_1, L_2	instruction labels
D_C, D_M	class and method declarations (respectively)
T	Java types
S	statements

$$D_C ::= \texttt{class}\ C_1\ \texttt{extends}\ C_2\ \{\ \overline{T\ f}\ \overline{D_M}\ \}$$

$$D_M ::= T\ m(\overline{T\ v})\ \{\ \overline{T\ v}\ \overline{S}\ \}$$

$$T ::= \texttt{int} \mid C$$

$$S ::= [L_1]\ v = e \mid [L_1]\ v.f = e \mid [L_1]\ v_1 = v_2.m(t_1, \ldots, t_n) \mid [L_1]\ \texttt{return}\ t \mid [L_1]\ \texttt{if}\ v\ bop\ t\ \texttt{goto}\ L_2 \mid [L_1]\ \texttt{goto}\ L_2$$

$$e ::= t_1\ aop\ t_2 \mid v.f \mid \texttt{new}\ C() \mid t$$

$$t ::= v \mid c$$

You may assume that local variables must be declared and defined before being used in a method. An example program in the *Bytecode Language* is:

```
class Test extends java.lang.Object {
     int aMethod(int j) {
          int i,Test p
          [0] i=0
          [1] if i>10 goto 5
          [2] i=i+1
          [3] j=j*2
          [4] goto 1
          [5] j = this.aMethod(j)
          [6] return j
     }
}
```

Assume the class hierarchy follows Java and `java.lang.Object` is the root. So, for example, `java.lang.Integer extends java.lang.Number`, etc.

Question 4. Code Generation [30 marks]

(a) [15 marks] Translate the following two Java programs into the *Bytecode Language* given on Page 4. You should attempt to produce Bytecode which is as efficient as possible.

(i)

```
class Test {
 int f(int x) {
  if(x > 10) { x = x + (x*2); }
  else { x = 1 + (2*x); }
  return x;
}}
```

(ii)

```
class Test {
 int f(int x, int y) {
  int r=0;
  for(int i=0;i<y;++i) { r += x*2; }
  return r;
}}
```

(b) [5 marks] An important problem which a compiler must solve is *register allocation*. Briefly, outline what this problem is about.

(c) [10 marks] The *live range* of each variable is used to construct an *interference graph*, which is then solved by *graph colouring*. Discuss what this means, and how it relates to register allocation.

Question 5. Dataflow Analysis [30 marks]

For this question, the *Bytecode Language* is modified so local variables are not declared. Thus, local variables may have *different* types at different points. *Bytecode verification* is needed to check types are used safely. For example, the following method is unsafe:

```
int f(int x) {
   [0] i = new java.lang.String(``Hello World'')
   [1] if x > 10 goto 3
   [2] i = new java.lang.Integer(0)
   [3] i = i.size()
   [4] return i
}
```

Here, `i` may reference an `Integer` object at instruction 3, but `Integer` has no `size()` method. The bytecode verifier determines `i` has type `Object` at that point, thus catching the problem.

(a) [5 marks] Draw a fragment of the *lattice of types* described in the *Bytecode Language*. Include at least the types: `Object`, `Number`, `Integer`, `Float`, `String` and `int`.

(b) [5 marks] Given your lattice, define the *least upper bound* of two types (denoted $T_1 \sqcup T_2$).

(c) [10 marks] Using the following description, give a suitable definition for the *least upper bound* of two *abstract stores* (denoted $S_1 \sqcup S_2$):

> The abstract store used by the bytecode verifier maps *variables* to *types*. If S is an abstract store, then $S[x]$ gives the type for variable x; likewise, $S[x \mapsto T]$ is an abstract store identical to S, except that x now has type T. Also, $dom(S)$ returns the variables mapped by S; so, $dom(\{x \mapsto T_1, y \mapsto T_2\}) = \{x, y\}$.

(d) [10 marks] Complete the **five** cases marked with **???** in the description of the *transfer function* given below:

> The *transfer function* used in the bytecode verifier *propagates type information*, but does not *check for correctness* (this is done separately). A partial definition of it is given as follows:
>
> | | | | | | |
> |---|---|---|---|---|---|
> | $f(v = e, S)$ | $=$ | $S[v \mapsto eval(e, S)]$ | $eval(c, S)$ | $=$ | **???** |
> | $f(v.f = e, S)$ | $=$ | **???** | $eval(v, S)$ | $=$ | **???** |
> | $f(v_1 = v_2.m(t_1, \ldots, t_2), S)$ | $=$ | $S[v_1 \mapsto rtype(m, S[v_2])]$ | $eval(v.f, S)$ | $=$ | **???** |
> | $f(\texttt{goto } L, S)$ | $=$ | S | $eval(\texttt{new } C(), S)$ | $=$ | C |
> | $f(\texttt{if } v \textit{ bop } t \texttt{ goto } L, S)$ | $=$ | **???** | $eval(t_1 \textit{ aop } t_2, S)$ | $=$ | `int` |
> | $f(\texttt{return } v, S)$ | $=$ | $\bot$ | | | |
>
> The function $rtype(m, c)$ gives the return type for method m in class c. A similar function, $ftype(f, c)$, gives the type of a field f in class c.

Question 6. Constraint-Based Analysis [30 marks]

You are writing a *type analysis* for use in an optimising Java compiler. The aim of the analysis is to identify method calls that dispatch to a single method.

In the analysis, a constraint of the form $X \supseteq \{\texttt{java.lang.Integer}\}$ indicates the program variable represented by X may refer to an instance of `java.lang.Integer` in some execution(s) of the program. The analysis uses the following constraint resolution rule:

$$C \cup \{X \supseteq Y\} \cup \{Y \supseteq Z\} \Longrightarrow C \cup \{X \supseteq Y\} \cup \{Y \supseteq Z\} \cup \{X \supseteq Z\} \quad (1)$$

Here, X, Y and Z are *constraint variables*, whilst C is the *constraint set* being solved.

The type analysis uses one constraint variable to model *every instance* of a particular local variable/parameter (i.e. it is *flow-insensitive*); it also uses one constraint variable to model *every instance* of a particular field (i.e. it is *field-insensitive*).

An example program in the Bytecode Language from Page 4 is:

```
class Test extends java.lang.Object {
     Test a

     void f() {
         Test t1, Test t2
         [0] t1 = new Test()
         [1] this.a = t1
         [2] t1 = new Test()
         [3] t1.a = new Test2()
         [4] t2 = this.a
         [5] t2.f()
     }
}

class Test2 extends Test {...}
```

(a) [10 marks] Write out the *initial* constraint set that would be generated from the above program. You should identify which statement each constraint comes from.

(b) [5 marks] Write out the *final* constraint set that would be generated from your initial constraint set using the resolution rule given above.

(c) [10 marks] Discuss the ways in the which the type analysis described above can be *imprecise*.

(d) [5 marks] Before the analysis is run, a conservative representation of the program's call graph must be constructed. Outline a simple way of constructing a program's call graph when the type analysis solution is not yet available.

****