Solutions

1. Short Answers [25 pts]

(a) [2 pts] A bottom-up parser might yield a reduce-reduce conflict while processing an input program with incorrect syntax. True or false?

*False, reduce-reduce errors might occur when constructing the parser, not when running it.*

(b) [2 pts] In a language with a sound type system, evaluation of expressions never gets stuck. True or false?

*True*

(c) [2 pts] Structural operational semantics defines the meaning of a program by induction on the structure of the program's type. True or false?

*False, it’s by induction on the structure of the program expression.*

(d) [3 pts] Which of the following analyses have distributive transfer functions?

- Variable liveness
- Available expressions
- Constant folding
- Reaching definitions

*All, except constant folding.*

(e) [3 pts] If types A, B, and C are three types such that A is a subtype of B, and B is a subtype of C, then which of the following function types are subtypes of B → B?

1) A → B  2) B → C  3) C → C  4) C → A  5) A → A

*Only 4, because the parameter type must be contravariant (B or C) and the return type must be covariant (A or B).*

(f) [2 pts] Consider the following SML program which declares a function f with a nested function g. Is the function f tail-recursive?

```sml
fun f(n:int, g2: int->int) : int =
  let fun g(y: int) = y + n + 10
  in
    if n = 1 then n + g(n) + g2(n)
    else f(n-1, g)
  end
end
```

*True—it is tail-recursive.*

(g) [2 pts] Now suppose we also define a function g0:

```sml
fun g0(y: int) : int = 100
```

What is the value of the expression f(1, g0), using the definition of f from part 1(f)? Recall that SML has lexical variable scoping.

*113*

(h) [2 pts] What is the value of the expression f(2, g0)?

*26. With dynamic scoping, it would be 25.*

(i) [3 pts] What does the following Prolog predicate f do? (Recall that [X|Y] is the Prolog notation for list cons, similar to ML’s X :: Y.)
It determines whether a list is a prefix of another list.

(j) [2 pts] Reference counting can fail to identify some garbage. True or false?

True, if there are cycles.

(k) [2 pts] If the Hoare triple \( \{P\}c\{Q\} \) holds, then so does \( \{P \lor P'\}c\{Q\} \) for any formula \( P' \). True or false?

False.

2. Object-Oriented Features [25 pts]

Consider an object-oriented language similar to Java with the following syntax:

Classes: \( C ::= \text{class } c \text{ extends } c' \{ c \bar{f}; \bar{d} \} \)

Methods: \( d ::= c m(\bar{c}\bar{f}) \{ \text{return } e; \} \)

Expressions: \( e ::= x \mid \text{new } c(\bar{e}) \mid \text{null} \mid \text{this} \mid e.f \mid e.m(\bar{e}) \)

where \( c \) ranges over class names, \( f \) over field names, \( m \) over method names, and \( x \) over variables. We use overlines as a shorthand for sequences. For instance, \( \bar{c} \bar{f} \) represents a sequence of field declarations \( c_1 f_1; \ldots; c_n f_n \). All of the methods in the program are virtual. Program variables \( x \) include the special variable \( \text{this} \) that represents the receiver object. Class names include a special class \( \text{Object} \) with no fields or methods. The expression \( \text{new } c(\bar{e}) \) creates a new object of class \( c \) and initializes its fields \( \bar{f} \) with the values of \( \bar{e} \). Each class name can be used throughout the program, even before the class is defined. A program consists of a set of class declarations and an expression \( e \) to evaluate.

The above language supports method overloading and overriding. This makes it possible to declare multiple methods with the same name in the same class hierarchy, as illustrated in the example below:

```
class A extends Object {
    B f;
    A m(A a) { return a; } /* method 1 */
    A m(B b) { return b; } /* method 2 */
}
class B extends A {
    A m(A a) { return a.f; } /* method 3 */
    A m(B b) { return b.f; } /* method 4 */
}
```

(a) [5 pts] Define the concepts of method overriding and method overloading, and indicate where they occur in the example above.

**Answer:**

Overloading: two methods of a class are overloaded when they have the same name, but different numbers or types of parameters. The methods do not necessarily have to be defined in the same class. Method 1 and 2 are overloaded; methods 3 and 4 are also overloaded.

Overriding: method \( m \) overrides method \( p \) if they have the same name, the same number of parameters, same parameter and return types, and \( m \)'s class is a subclass of \( p \)'s class. Method 3 overrides method 1; and method 4 overrides method 2.

(b) [3 pts] Briefly explain (in no more than 2 sentences) when and how are method overloading and overriding being resolved?

**Answer:**

Overloaded methods are resolved statically, based on method signatures. Overiden methods are resolved dynamically, based on the run-time type of the receiver object.
(c) [7 pts] Now we want to define a model that precisely describes the semantics of our language. We describe the evaluation of expressions using a small-step evaluation relation \( e \rightarrow e' \). The evaluation eventually yields a value \( v \), which is either an object or null:

\[
\text{Values } v ::= \text{null } | \text{new } c(v)
\]

The following rules describe the execution of all expressions except method calls:

\[
\begin{align*}
\text{new } c(v) & \rightarrow \text{new } c(v') \\
\text{e} & \rightarrow e' \\
f_i \in \text{fields}(c) & \quad v_i \in v
\end{align*}
\]

Here, \( \text{fields}(c) \) is the sequence of fields of class \( c \), including those defined in the superclasses. The notation \( v_i \in v \) indicates that \( v_i \) is the \( i \)-th element in the sequence \( v \) (and similarly for \( f_i \)). The evaluation of sequences \( \bar{v} \rightarrow \bar{v'} \) is done by evaluating each expression from left to right:

\[
e_i \rightarrow e_i' \\
(v_1, \ldots, v_{i-1}, e_i, e_{i+1}, \ldots, e_n) \rightarrow (v_1, \ldots, v_{i-1}, e_i', e_{i+1}, \ldots, e_n)
\]

For method calls, our semantic model first records the static types of its arguments, before evaluating the call:

\[
\forall e_i \in \bar{e}, c_i \in \bar{e} : c_i = \text{type}(e_i)
\]

\[
e.m(\bar{e}) \rightarrow e.m(\bar{e'})
\]

Write the remaining rules to complete the evaluation of method calls in the presence of overloading and overriding. You can assume that \( \text{decl}(c, m, \bar{c}) \) returns the declaration of the method with name \( m \) and signature \( \bar{c} \) in class \( c \).

Answer:

\[
\begin{align*}
e & \rightarrow e' \\
\text{e.m}(\bar{e})(\bar{c}) & \rightarrow \text{e'.m}(\bar{e})(\bar{c}) \\
v.m(\bar{v})(\bar{c}) & \rightarrow v.m(\bar{v})(\bar{c'})
\end{align*}
\]

\[
\text{new } c(v).m(\bar{c})(\bar{v'}) \rightarrow e\{ \bar{x} \mapsto \bar{v'} \}\{ \text{this } \mapsto \text{new } c(v) \}
\]

(d) [6 pts] Next, we are concerned with the implementation of our language in a standard virtual machine. Consider a program that evaluates the expression:

\[
(\text{new } A(\text{null})).m(\text{new } A(\text{new } B(\text{null})))
\]

where classes \( A \) and \( B \) are the ones defined at the beginning of the problem. Draw a reasonable memory layout of the virtual machine when the program execution is about to evaluate the body of method \( m \). Clearly indicate the stack, the heap, and the static area in your figure.
(e) [4 pts] In C++, the usual object-oriented method behavior is obtained by defining methods with the keyword `virtual`. By default, C++ methods (i.e., without the `virtual` or `static` qualifiers) are not virtual, but are still able to access the fields of the enclosing object. Briefly compare the way virtual methods and C++ default methods are implemented, indicating how they differ and in what respects they are similar. (2–3 sentences)

**Answer:**

Virtual methods are compiled into indirect calls that use vtable lookups, while default methods are compiled into direct calls to methods determined by the static type of the receiver object. However, both virtual and default methods require an extra parameter for the receiver object, to be able to access object fields.

3. **Dataflow analysis [25 pts]**

Consider the control-flow graph $G = (V, E)$ for a procedure, which has a distinguished node `START` that has the property that there is a path from `START` to every node in $V$. A node $d$ is said to dominate a node $v$ if every path from `START` to $v$ contains $d$.

(a) [5 pts] Draw a CFG for the following IMP code, with each node annotated by the corresponding line number:

1: if $x > 0$ then
2: while $n < 10$ do
3: $n := n + x$
4: else
5: $n := 0$

**Answer:**

```
START
1: if $x > 0$
2: if $n < 0$
3: $n := n + x$
5: $n := 0$
END
```

(b) [5 pts] The dominance relation is the set of pairs $(d, v)$ such that $d$ dominates $v$. Give the dominance relation for the previous example.
(c) [5 pts] Write down a set of dataflow equations for computing the dominance relation. Your answer must state clearly the domain for equations and whether your dataflow scheme is forward-flow or backward-flow.

Answer:

Domain of equations: powerset of V, ordered by inclusion. Least element is empty set.
Forward-flow, all paths problem
Equations:

- at a node n, Out(n) = \{n\} ∪ In(n)
- confluence operator is intersection

(d) [5 pts] A program is said to be in static single assignment (SSA) form if every variable is defined exactly once in the program and every use of a variable is reached by exactly one definition of that variable in the program. By convention, START is considered to be a (second) definition for all variables in the program. Assuming every variable is known to be defined once, briefly explain how you would use a standard dataflow analysis to determine whether or not a program is in SSA form.

Answer:

Solve the standard reaching definitions problem and for each use of a variable in the program, check that only its one real definition reaches it.

(e) [5 pts] Suppose a program is in SSA form. If a definition d of a variable x is the only definition that reaches a use of x, is it necessarily true that d dominates that use of x?

Answer:

Yes. If not, there is a path P from START to that use that does not contain d. So either START or some other node on P will contain a definition that reaches the use, which contradicts the assumption that only one definition reaches this use.

4. Types and Exceptions [25 pts]

Exceptions are a commonly supported language feature, but they are supported differently in different languages. In SML, exceptions are not statically checked, so statically the type checker does not know what exceptions a given function might throw. In Java, (most) exceptions must be statically declared and the type checker ensures that all exceptions are caught unless they are allowed to escape.

(a) [5 pts] Why are exceptions a useful language feature? Discuss briefly in two or three sentences.

Answer:

Exceptions permit exceptional cases (such as errors) to be handled by code that is cleanly separated from the normal case code. Further, when exceptional cases are forgotten, code is relatively cleanly terminated rather than continuing and computing garbage. The alternatives are worse: encoding error values into the space of the normal result type of a function (when possible) invites ignoring errors; returning a record or variant with added information to signal exceptions tends to clutter the code with handling of exceptions.

Let us explore static typing of exceptions in the context of the simply typed lambda calculus. The typed lambda calculus with integers has the following syntax.

\[
x \in \text{Var} \\
n \in \mathbb{Z} \\
e ::= n \mid x \mid \lambda x: \tau. e \mid e_0 e_1
\]
A program is a well-typed term $e$, which evaluates according to a standard small-step operational semantics $e \rightarrow e'$ to arrive at a value $v$, which is either an integer $n$ or a lambda term $\lambda x : \tau. e$.

The following new terms generate and handle exceptions respectively:

$$e ::= \ldots \mid \text{throw } X \mid \text{try } e_1 \text{ catch } X \Rightarrow e_2$$

In the throw expression, $X$ is the name of the exception. For simplicity, there is no associated value as there would be in SML or Java. Informally, the try expression evaluates $e_1$ to obtain the value of the whole expression, but if $e_1$ throws the exception $x$, the expression $e_2$ is evaluated instead. Throwing an exception $X$ immediately transfers control to the closest dynamically enclosing clause catch$(X)$ with the same name $X$.

To statically keep track of possible exceptions, the typing judgement will take the form

$$\Gamma \vdash e : \tau, S$$

where $S$ is a set $\{X_1, \ldots, X_n\}$ including all exceptions that might be thrown by $e$. Further, the type of a function must, as in Java, include its possible exceptions, so it will have the form $\tau \rightarrow \tau'/S$ meaning that it either has a result of type $\tau'$ or throws one of the exceptions in the set $S$.

(b) [12 pts] Complete the typing rules below for this language.

\begin{align*}
\tau & ::= \text{int} \mid \tau \rightarrow \tau' / S \\
\Gamma & ::= \Gamma, x : \tau \mid \emptyset \\
\Gamma \vdash n : \text{int}, \emptyset & \quad x \in \text{dom}(\Gamma) \\
\Gamma \vdash x : \Gamma(x), \emptyset & \\
\Gamma \vdash e_0 : \tau \rightarrow \tau'/S_2, S_0 & \quad \Gamma \vdash e_1 : \tau, S_1 \\
\Gamma \vdash e_0 e_1 : \tau', S_0 \cup S_1 \cup S_2 & \\
\Gamma \vdash (\lambda x : \tau. e) : \tau \rightarrow \tau'/S', \emptyset & \\
\Gamma \vdash \text{throw } X : \tau, \{X\} \\
\Gamma \vdash e_1 : \tau, S_1 & \quad \Gamma \vdash e_2 : \tau, S_2 \\
\Gamma \vdash \text{try } e_1 \text{ catch } X \Rightarrow e_2 : \tau, (S_1 - \{X\}) \cup S_2
\end{align*}

(c) [3 pts] What guarantees that a type checker for this language, implemented according to these rules, will terminate? Explain in one or two sentences.

**Answer:**

A recursive type checker for this language will terminate because in each rule, the expressions appearing in the antecedent are proper subexpressions of the expression in the premise. Therefore, the type checker can only recurse to finite depth. Further, there are only a finite number of premises in every rule, so the proof tree explored by the type checker has finite size.

(d) [5 pts] Suppose we wanted to translate this language into a target language that is the simply typed lambda calculus with sum types $\tau_1 + \tau_2$. If a term has type $\tau, \{X_1, \ldots, X_n\}$, what target language type might be used as a suitable representation? Sketch how this representation would work in 2–3 sentences.

**Answer:**

One reasonable choice would be the type $\tau + \text{int}$. The int case would be used to represent an exceptional result, with the integer encoding which of $X_1, \ldots, X_n$ was thrown. Every use of an expression with type $\tau$ is translated to code that evaluates the expression and then checks whether its result was an exception, possibly remapping the integer index.