Problem 1 (50pts)

In this problem we will study a simplified version of the try...catch...finally construct of Java. Consider a language consisting of

- A set Exc of *exceptions* $E, F, \ldots$
- A set of *atomic statements* a
- A set of *tests* b
- A set Stmt of *statements* $c, d, e, \ldots$ given by the following grammar:

$$\begin{array}{rcl} c & ::= & a \;\mid\; \mathsf{skip} \;\mid\; \mathsf{throw}\,E \;\mid\; c\,;\,d \;\mid\; \mathsf{if}\,(b)\,\{c\}\,\mathsf{else}\,\{d\} \\ & \mid & \mathsf{try}\,\{c\}\,\mathsf{catch}\,E\,\{d\}\,\mathsf{finally}\,\{e\} \end{array}$$

Either the catch clause or the finally clause may be omitted in the try statement.

Intuitively, $\mathsf{try}\,\{c\}\,\mathsf{catch}\,E\,\{d\}\,\mathsf{finally}\,\{e\}$ works as follows. Statements may terminate normally or may terminate exceptionally by throwing an uncaught exception. The statement d is called an *exception handler* for E, and its scope is c. It is invoked if c throws an uncaught exception E. If c terminates normally, or if c throws an exception $F \neq E$, then d is never invoked.

The finally clause e is always executed last, regardless of whether c or d terminate normally or exceptionally. Any uncaught exception other than E thrown by c or any uncaught exception thrown by d is rethrown upon normal termination of e. If e throws an exception F, then e terminates exceptionally and F is rethrown. In all other cases, the try statement terminates normally.

Assume that with each atomic statement a there is associated a total function $M_a : \mathsf{S} \to \mathsf{S}$ on some set of *states* S, and with each test b there is associated a total Boolean valued function $M_b : \mathsf{S} \to \mathsf{Bool}$. The exact nature of S, M_a, and M_b are unimportant.

(a) [7pts] We wish to define a large-step operational semantics for this language. Programs will be interpreted as partial functions $\mathcal{D} \rightharpoonup \mathcal{D}$, where $\mathcal{D} = \mathsf{S} + (\mathsf{Exc} \times \mathsf{S})$. We assume that S and $\mathsf{Exc} \times \mathsf{S}$ are disjoint and dispense with injection functions to simplify notation. Symbols $s, t, u, \ldots$ denote elements of S and $T, U, V, \ldots$ denote elements of $\mathcal{D}$. An output of the form (E, s) means that exception E has been thrown in state s; an output of the form s means that the program terminated normally in state s. The notation $\langle c, s\rangle \to T$ means that if statement c is started in state s then it halts in output state T.

Here are the semantic rules for all constructs except the try statement:

$$\langle \mathsf{skip}, s\rangle \to s \qquad \frac{\langle c, s\rangle \to T \qquad \langle d, T\rangle \to U}{\langle c\,;\,d, s\rangle \to U}$$

$$\langle a, s\rangle \to M_a(s) \qquad \frac{\langle c_1, s\rangle \to T \qquad M_b(s)}{\langle \mathsf{if}\,(b)\,\{c_1\}\,\mathsf{else}\,\{c_2\}, s\rangle \to T}$$

$$\langle c, (E,t)\rangle \to (E,t)$$

$$\langle \mathsf{throw}\,E, s\rangle \to (E,s) \qquad \frac{\langle c_2, s\rangle \to T \qquad \neg M_b(s)}{\langle \mathsf{if}\,(b)\,\{c_1\}\,\mathsf{else}\,\{c_2\}, s\rangle \to T}$$

Here are the rules for the try statement with either the catch clause or the $\mathsf{finally}$ clause omitted:

$$\frac{\texttt{<}c\,,s\texttt{>} \to t}{\texttt{<}\mathsf{try}\,\{c\}\,\mathsf{catch}\,E\,\{d\}\,,s\texttt{>} \to t} \qquad \frac{\texttt{<}c\,,s\texttt{>} \to t \qquad \texttt{<}e\,,t\texttt{>} \to U}{\texttt{<}\mathsf{try}\,\{c\}\,\mathsf{finally}\,\{e\}\,,s\texttt{>} \to U}$$

$$\frac{\texttt{<}c\,,s\texttt{>} \to (E,t) \qquad \texttt{<}d\,,t\texttt{>} \to U}{\texttt{<}\mathsf{try}\,\{c\}\,\mathsf{catch}\,E\,\{d\}\,,s\texttt{>} \to U} \qquad \frac{\texttt{<}c\,,s\texttt{>} \to (F,t) \qquad \texttt{<}e\,;\,\mathsf{throw}\,F\,,t\texttt{>} \to U}{\texttt{<}\mathsf{try}\,\{c\}\,\mathsf{finally}\,\{e\}\,,s\texttt{>} \to U}$$

$$\frac{\texttt{<}c\,,s\texttt{>} \to (F,t) \qquad F \neq E}{\texttt{<}\mathsf{try}\,\{c\}\,\mathsf{catch}\,E\,\{d\}\,,s\texttt{>} \to (F,t)}$$

Give rules for the full try statement. There should be four.

(b) [6pts] As mentioned, we may omit either the catch clause or the $\mathsf{finally}$ clause of the try statement. However, these constructs are redundant, since we can define them in terms of the full try statement. Conversely, we can define $\mathsf{try}...\mathsf{catch}...\mathsf{finally}$ in terms of $\mathsf{try}...\mathsf{catch}$ and $\mathsf{try}...\mathsf{finally}$. Give these three definitions.

(c) [7pts] Java allows try statements with multiple handlers, e.g.

$$\mathsf{try}\,\{c\}\,\mathsf{catch}\,E_1\,\{d_1\}\,\mathsf{catch}\,E_2\,\{d_2\}$$

where $E_1 \neq E_2$. The scope of both handlers is c. Show that this construct is redundant by defining it in terms of the single-handler try statement.

Note that the answer

$$\mathsf{try}\,\{\mathsf{try}\,\{c\}\,\mathsf{catch}\,E_1\,\{d_1\}\}\,\mathsf{catch}\,E_2\,\{d_2\}$$

is incorrect, because an exception E_2 thrown by d_1 would be erroneously caught by d_2. Hint: use a new exception not occurring in the statement.

(d) [6pts] Show that $\mathsf{finally}$ is redundant in the presence of $\mathsf{try}...\mathsf{catch}$. That is, show how to define the construct $\mathsf{try}\,\{c\}\,\mathsf{catch}\,E\,\{d\}\,\mathsf{finally}\,\{e\}$ in terms of the construct $\mathsf{try}\,\{c\}\,\mathsf{catch}\,E\,\{d\}$. You may assume that your solution to the previous problem has been generalized to arbitrarily many handlers, and you may assume knowledge of all the exceptions that can be thrown by c.

Now we will formulate a continuation-passing semantics involving two continuations, a normal continuation $N : \mathsf{S} \rightharpoonup \mathsf{S}$ and an exceptional continuation $X : \mathsf{Exc} \to \mathsf{S} \rightharpoonup \mathsf{S}$ to be invoked upon normal and exceptional termination, respectively. Define

$$\mathsf{NCont} \stackrel{\text{def}}{=} \mathsf{S} \rightharpoonup \mathsf{S} \qquad \mathsf{XCont} \stackrel{\text{def}}{=} \mathsf{Exc} \to \mathsf{S} \rightharpoonup \mathsf{S}$$

The meaning function is

$$[\![\ ]\!] : \mathsf{Stmt} \to \mathsf{NCont} \to \mathsf{XCont} \to \mathsf{S} \rightharpoonup \mathsf{S}.$$

The meanings of all statements except the try statement are defined as follows.

$$\begin{aligned}
[\![a]\!]NX &\stackrel{\text{def}}{=} N \circ M_a \\
[\![\mathsf{skip}]\!]NX &\stackrel{\text{def}}{=} N \\
[\![\mathsf{throw}\,E]\!]NX &\stackrel{\text{def}}{=} XE \\
[\![c_1\,;\,c_2]\!]NX &\stackrel{\text{def}}{=} [\![c_1]\!]([\![c_2]\!]NX)X \\
[\![\mathsf{if}\,(b)\,\{c_1\}\,\mathsf{else}\,\{c_2\}]\!]NX &\stackrel{\text{def}}{=} \lambda s.\text{if } M_b(s) \text{ then } [\![c_1]\!]NXs \text{ else } [\![c_2]\!]NXs
\end{aligned}$$

The meanings of the try statement with either the catch or the $\mathsf{finally}$ clause omitted can be defined as follows:

$$\begin{aligned}
[\![\mathsf{try}\,\{c\}\,\mathsf{catch}\,E\,\{d\}]\!]NX &\stackrel{\text{def}}{=} [\![c]\!]N(\lambda F.\text{if } F = E \text{ then } [\![d]\!]NX \text{ else } XF) \\
[\![\mathsf{try}\,\{c\}\,\mathsf{finally}\,\{e\}]\!]NX &\stackrel{\text{def}}{=} [\![c]\!]([\![e]\!]NX)(\lambda F.[\![e]\!](XF)X)
\end{aligned}$$

(e) [6pts] Give a definition of the full try statement.

Now we wish to add function calls to the language. A *program* now consists of a finite sequence of function declarations

$$\mathsf{function}\, f\, \mathsf{throws}\, E_1, \ldots, E_n\, \{c\}$$

(at most one for each function name f) followed by a statement. We also add a new atomic statement $\mathsf{call}\, f$. The declared functions may be mutually recursive. For example,

$$\begin{array}{l} \mathsf{function}\, f\, \mathsf{throws}\, E\, \{\mathsf{if}\ (b_1)\ \{\mathsf{throw}\, E\}\ \mathsf{else}\ \{a_1\ ;\ \mathsf{call}\, g\}\}; \\ \mathsf{function}\, g\, \mathsf{throws}\, E\, \{\mathsf{if}\ (b_2)\ \{\mathsf{throw}\, E\}\ \mathsf{else}\ \{a_2\ ;\ \mathsf{call}\, f\}\}; \\ \mathsf{try}\ \{\mathsf{call}\, f\}\ \mathsf{catch}\, E\ \{a_3\} \end{array}$$

Any uncaught exception that could be thrown by the body of a function must be declared in the function declaration. If the body of a function throws an uncaught exception, then the function terminates exceptionally and the same exception is rethrown at the call site.

(f) [6pts] Give the continuation-passing semantics of $\mathsf{call}\, f$, and explain what needs to be done to ensure that the semantics is well-defined when the declared functions are mutually recursive.

(g) [7pts] Design a type system for proving that all possible uncaught exceptions thrown by the body of a function are declared in the function declaration. Your type system should have judgements of the form

$$\vdash c\ \mathsf{throws}\ \Delta \qquad \vdash f\ \mathsf{is\ ok}$$

where c is a statement, Δ is a set of exceptions, and f is a declared function. Intuitively, $\vdash c\ \mathsf{throws}\ \Delta$ means that all uncaught exceptions that can be thrown by c are in Δ, and $\vdash f\ \mathsf{is\ ok}$ means that the body of f does not throw any undeclared exceptions. One rule of your system will be

$$\frac{\vdash c\ \mathsf{throws}\ \Delta \qquad \Delta \subseteq \{E_1, \ldots, E_n\}}{\vdash f\ \mathsf{is\ ok}}$$

where the declaration of f is $\mathsf{function}\, f\, \mathsf{throws}\, E_1, \ldots, E_n\, \{c\}$.

(h) [5pts] Explain how the explicit declaration of thrown exceptions simplifies the previous problem.

Problem 2 (50pts)

Consider a simple object-oriented language with the following syntax:

$$\begin{array}{rcl} P & ::= & d*\ s \\ d & ::= & \mathsf{class}\ c\ \mathsf{extends}\ c'\ \{ \\ & & \quad t\ f; \\ & & \quad m(t\ x)\ \{s\} \\ & & \} \\ t & ::= & c\ \mid\ \mathsf{int}\ \mid\ \mathsf{bool} \end{array} \qquad \begin{array}{rcl} c & \in & C \text{ (class names)} \\ f & \in & F \text{ (field names)} \\ m & \in & M \text{ (method names)} \\ x, y, z & \in & V \text{ (variable names)} \end{array}$$

In this language, classes extend their superclasses with exactly one field f and one method m. Each method takes exactly one argument x and returns no value, being executed only for its side-effects. The set C of classes contains a base class Object, whose definition is not relevant and is omitted. The statements have the following syntax:

$$s \ ::= \ s_0; s_1 \ \mid\ t\ x\ \mid\ \mathsf{if}\ x\ \mathsf{then}\ s_0\ \mathsf{else}\ s_1 \ \mid\ x.m(y)\ \mid\ \mathit{assign}$$

Hence, statements are sequences of variable declarations, if constructs, method calls, and assignments. The call statement $x.m(y)$ invokes the method m of the object referenced by x, passing y as parameter. Each assignment can be one of the following three-address statements:

$$\begin{array}{rcl} \mathit{assign} & ::= & x = v \ \mid\ x = y \ \mid\ x = y\ \mathit{op}\ z \ \mid\ x = \mathsf{new}\ c \ \mid\ x = \mathsf{null} \ \mid\ x = y.f \ \mid\ x.f = y \\ \mathit{op} & ::= & + \ \mid\ - \ \mid\ \leq \\ v & ::= & n \ \mid\ \mathsf{true} \ \mid\ \mathsf{false} \end{array}$$

These statements manipulate integers, booleans, and object references. The statements $x = y$, $x = y.f$ and $x.f = y$ assign variable values or object field values to other variables or object fields. The statement $x = \textsf{new}\ c$ creates a new object c and stores a reference to it in x. By default, all of the reference fields in the newly created object have null values. In these statements, $x, y, z \in V$ represent local variables or method arguments, and are allocated on the stack. We will therefore refer to such variables as stack variables.

(a) [6pts] This language supports only single inheritance. Other languages like C++ support multiple inheritance. What is a standard complication caused by multiple inheritance?

(b) [6pts] In this language, method calls are dynamically dispatched. Consider the following program:

```
class A { int m(int x) { ...} }
class B extends A { int m(int x) { ...} }

A o; int i; bool b;
if (b) then o = new B;
       else  o = new A;
o.m(i);
```

In this example, the method call o.m(i) can invoke either A's method m or B's method m, depending on which branch of the if statement is executed.

Object-oriented languages use a run-time mechanism to ensure that the program dynamically dispatches the appropriate method. Briefly describe the functioning of this run-time system and specify whether its data structures are created statically or dynamically.

The mechanism for dynamically dispatched calls introduces run-time overhead and slows down the program. To reduce this overhead, the compiler can optimize the program by generating direct calls whenever it can precisely determine the class of the receiver object. We therefore want to design a dataflow analysis, called Class Hierarchy Analysis, which computes, for each object reference x, a set S of possible classes for the object referenced by x. The compiler can then replace the method call $x.m()$ with a direct call whenever there is a unique method m for all of the classes in S.

The Class Hierarchy Analysis keeps track the possible classes for each stack variable $x \in V$. However, for statements of the form $x = y.f$ the analysis also needs class information for the field f of the heap-allocated object referenced by y. We therefore build our dataflow abstraction as a pair:

$$\begin{array}{rcl} (s, h) & & \text{where:} \\ s & : & V \to \mathcal{P}(C \cup \{\textsf{null}\}) \\ h & : & (C \times F) \to \mathcal{P}(C \cup \{\textsf{null}\}) \end{array}$$

The s (stack) component models the possible classes of objects referenced by stack variables: if $x \in V$, then the set $s(x)$ conservatively approximates the classes of all objects that x may reference. The h (heap) component models the possible classes of object fields: if o is an object created by $\textsf{new}\ c$, then the set $h(c, f)$ approximates the classes of the objects that $o.f$ may reference.

(c) [6pts] In our analysis, the smaller the sets of classes $s(x)$ and $h(c, f)$, the more precise the analysis results. We want to construct a complete lattice over our analysis domain, such that elements higher up in the lattice represent more precise analysis information. Thus, the lattice should have top element $(\lambda x.\varnothing,\ \lambda(c, f).\varnothing)$. What is the partial order relation in this lattice?

(d) [6pts] To simplify the presentation of transfer functions we use the following notation. Given a function f, $f' = f[S \mapsto T]$ maps each element in S to T, and $f'' = f[S \overset{\oplus}{\mapsto} T]$ adds T to the set of values of each element in S:

$$f'(x) = \begin{cases} T & \text{if } x \in S \\ f(x) & \text{if } x \notin S \end{cases} \qquad\qquad f''(x) = \begin{cases} f(x) \cup T & \text{if } x \in S \\ f(x) & \text{if } x \notin S \end{cases}$$

If $\mathit{fields}(c)$ is the set of all fields of class c, the transfer function for $x = \mathsf{new}\ c$ is:

$$[\![x = \mathsf{new}\ c]\!]\ (s,h) \quad = \quad (\ s[\ \{x\} \mapsto \{c\}\ ],\ h[\ \{(c,f) \mid f \in \mathit{fields}(c)\} \overset{\oplus}{\mapsto} \{\mathsf{null}\}\ ]\ )$$

In the stack component s, the analysis updates the value of x, killing its old value: $\{x\} \mapsto \{c\}$. However, in the heap component h, the analysis adds null to the existing information, without killing the old values: $\{\ldots\} \overset{\oplus}{\mapsto} \{\mathsf{null}\}$. Why is it safe to kill existing values in the stack component, but not in the heap component?

(e) [7pts] Other transfer functions are:

$$[\![x = \mathsf{null}]\!]\ (s,h) \quad = \quad (\ s[\ \{x\} \mapsto \{\mathsf{null}\}\ ],\ h\ )$$

$$[\![x = y.f]\!]\ (s,h) \quad = \quad (\ s[\ \{x\} \mapsto \bigcup_{z \in s(y) - \{\mathsf{null}\}} h(z,f)\ ],\ h\ )$$

Write the transfer functions for $x = y$ and $x.f = y$.

(f) [6pts] Consider now the method invocation statements $x.m(y)$. In the worst case, such a call can arbitrarily update each field of each object with arbitrary object references. Write the transfer function for $x.m(y)$ which safely characterizes this worst case scenario.

(g) [7pts] A more precise approach is to perform an inter-procedural analysis: for each call $x.m(y)$, analyze the body of the invoked method m. However, we don't know statically which particular method m refers to (that's why we started designing the analysis in the first place). The solution is to consider the set of all possibly invoked methods, analyze each of them separately, then combine the results. For each method, the analysis must bind its formal argument to the actual value y at the call site before analyzing the body of the method.

In the absence of any information about x, we would have to analyze all methods named m in the program. Luckily, the analysis already has information about x in $s(x)$; we can therefore use this information and analyze only the methods m that occur in classes in $s(x)$.

Write the transfer function for $x.m(y)$ described above. Use $\mathit{meth}(c)$ for the set of all methods in c, $\arg(m,c)$ for the formal argument of method m of class c, and $\mathit{body}(m,c)$ for the body of method m of class c.

(h) [6pts] What properties of the dataflow abstraction ensure that the analysis always terminates, even when the program creates an unbounded number of objects?