Instructions

• This is a closed-book exam.

• You have 120 minutes to answer all of the questions. The entire exam is worth 120 points.

• Questions vary significantly in difficulty, and the point value of a given question is not always exactly proportional to its difficulty. Do not spend too much time on any one question.

• Partial credit will be given. All correct answers are short. The back side of each page and the companion handout may be used as scratch paper.

• Good luck!
Inductively Defined Relations

Define the syntactic categories of *blobs* (written $x$) and *counts* (written $y$) as follows:

$$
x ::= \# \| b \| \natural \| x \cdot x
$$

$$
y ::= 0 \| +y \| -y
$$

That is, a blob is a tree whose leaves are labeled $\#$, $\natural$, or $b$; a count is a sequence of $+$s and $-$s ending in 0.

Now define the relation “accumulating $x$ onto $y$ yields $y'$,” written $x \triangleright y \triangleright y'$, as the least three-place relation closed under the following rules:

$$
\# \triangleright y \triangleright +y \quad \text{(Sharp)}
$$

$$
b \triangleright y \triangleright -y \quad \text{(Flat)}
$$

$$
\natural \triangleright y \triangleright y \quad \text{(Natural)}
$$

$$
\frac{x_1 \triangleright y \triangleright y'' \quad x_2 \triangleright y'' \triangleright y'}{x_1 \cdot x_2 \triangleright y \triangleright y'} \quad \text{(Dot)}
$$

$$
\frac{x_2 \cdot x_1 \triangleright y \triangleright y'}{x_1 \cdot x_2 \triangleright y \triangleright y'} \quad \text{(Swap)}
$$

Notice that the result of accumulating $x$ onto $y$ always has $y$ itself as a suffix, and that it additionally includes one $+$ for every $\#$ in $x$ and one $-$ for every $b$ in $x$. The middle component of the relation, $y$, is analogous to the “accumulator parameter” sometimes used by tail-recursive OCaml functions. The Swap rule introduces some flexibility in the order of $+$s and $-$s in $y'$, relative to the positions of $\#$s and $b$s in $x$.

1. (6 points) Are the following statements derivable? (Write YES or NO for each.)

   (a) $\# \cdot (\natural \cdot \sharp) \triangleright 0 \triangleright +0$

      *Answer:* Yes

   (b) $\# \cdot (\natural \cdot b) \triangleright 0 \triangleright + - 0$

      *Answer:* Yes

   (c) $(\natural \cdot \sharp) \cdot (b \cdot \natural) \triangleright + 0 \triangleright + + - 0$

      *Answer:* No

*Grading scheme:* Binary
2. (20 points) Write a careful inductive proof of the following fact. Make sure to explicitly mention every step in the proof (use of an assumption, use of the induction hypothesis, use of one of the inference rules, etc.).

**Fact:** For every $x$ there is some $y'$ such that $x \not\not\not\not 0 \not\not\not\not y'$

**Answer:** Proving the claim directly by induction on $x$ does not work. Instead, we prove the following, stronger fact: For every $x$ and $y$, there is some $y'$ such that $x \not\not\not\not y' \not\not\not\not y'$.

**Proof:** By induction on the structure of $x$.

- Case $x = \dagger$:
  - By **Sharp**, $\dagger \not\not\not\not y \not\not\not\not +y$ is derivable.
  - Thus, we can take $y'$ to be $+y$.
- Case $x = \flat$:
  - By **Flat**, $\flat \not\not\not\not y \not\not\not\not -y$ is derivable.
  - Thus, we can take $y'$ to be $-y$.
- Case $x = \natural$:
  - By **Natural**, $\natural \not\not\not\not y \not\not\not\not y$ is derivable.
  - Thus, we can take $y'$ to be $y$.
- Case $x = x_1 \cdot x_2$ (for some $x_1$ and $x_2$):
  - By the induction hypothesis for $x_1$, there is some $y''$ such that $x_1 \not\not\not\not y \not\not\not\not y''$.
  - By the induction hypothesis for $x_2$, there is some $y'''$ such that $x_2 \not\not\not\not y'' \not\not\not\not y'''$.
  - By **Dot**, we have that $(x_1 \cdot x_2) \not\not\not\not y \not\not\not\not y'''$.
  - Thus, we can take $y'$ to be $y'''$.

From our stronger fact, the desired fact immediately follows.

**Grading scheme:**

- -20 for performing induction on the structure of a derivation (rather than the structure of $x$)
- -10 for failing to strengthen the induction hypothesis (-5 for answers where the strengthened IH is given as a lemma but not proved)
  - In cases where an incorrect IH was used, 6 points (of the remaining 10) were allocated to the Sharp, Flat, and Natural cases; 4 for the Dot case
- -1 to -2 for small confusions.
Untyped Lambda-Calculus

The following problem concerns the untyped lambda-calculus. This system is summarized on page 1 of the companion handout.

3. (6 points) Recall the definitions of observational and behavioral equivalence from the lecture notes:

- Two terms $s$ and $t$ are **observationally equivalent** iff either both are normalizable (i.e., they reach a normal form after a finite number of evaluation steps) or both are divergent.
- Terms $s$ and $t$ are **behaviorally equivalent** iff, for every finite sequence of values $v_1, v_2, \ldots, v_n$ (including the empty sequence), the applications

  $$s \ v_1 \ v_2 \ \ldots \ v_n$$

  and

  $$t \ v_1 \ v_2 \ \ldots \ v_n$$

  are observationally equivalent.

For each of the following pairs of terms, write YES if the terms are behaviorally equivalent and NO if they are not.

(a) $$(\lambda x. \lambda y. x (\lambda z. z) y)$$
    and $$(\lambda x. \lambda y. (\lambda z. z) x y)$$
    Answer: NO

(b) $$(\lambda s. \lambda z. s (s z))$$
    and $$(\lambda n. \lambda s. \lambda z. s (n s z)) (\lambda s. \lambda z. s z)$$
    Answer: YES

(c) $$(\lambda x. x x) (\lambda x. x x)$$
    and $$(\lambda g. \lambda h. h) (\lambda z. z)$$
    where $Z = (\lambda f. \lambda y. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) y)$, as in lecture notes
    Answer: NO

Grading scheme: Binary.
Subtyping

The following problems concern the simply typed lambda-calculus with subtyping (and records, variants, and references). This system is summarized on page 2 of the companion handout.

4. (10 points) Circle T or F for each of the following statements.

(a) There is an infinite descending chain of distinct types in the subtype relation—that is, an infinite sequence of types $S_0$, $S_1$, etc., such that all the $S_i$’s are different and each $S_{i+1}$ is a subtype of $S_i$.

\[ \begin{array}{ll} T & F \\ \end{array} \]

(b) There is an infinite ascending chain of distinct types in the subtype relation—that is, an infinite sequence of types $S_0$, $S_1$, etc., such that all the $S_i$’s are different and each $S_{i+1}$ is a supertype of $S_i$.

\[ \begin{array}{ll} T & F \\ \end{array} \]

(c) There exists a type that is a subtype of every other type.

\[ \begin{array}{ll} T & F \\ \end{array} \]

(d) There exists a record type that is a subtype of every other record type.

\[ \begin{array}{ll} T & F \\ \end{array} \]

(e) There exists a variant type that is a subtype of every other variant type.

\[ \begin{array}{ll} T & F \\ \end{array} \]

Grading scheme: Binary
5. (15 points) The standard subtyping rule for references is:

\[
\frac{T_1 \leq S_1 \quad S_1 \leq T_1}{\text{Ref } S_1 \leq \text{Ref } T_1}
\]  

(S-Ref)

Suppose we drop the first premise so that Ref becomes a covariant type constructor:

\[
\frac{S_1 \leq T_1}{\text{Ref } S_1 \leq \text{Ref } T_1}
\]  

(S-Ref-New)

Indicate whether each of the following properties remains true (write “TRUE”) or becomes false (write “FALSE”), and briefly explain why.

(a) **Progress**: Suppose \( t \) is a closed, well-typed term (that is, \( \emptyset | \Sigma \vdash t : T \) for some \( T \) and \( \Sigma \)). Then either \( t \) is a value or else, for any store \( \mu \) such that \( \emptyset | \Sigma \vdash \mu \), there is some term \( t' \) and store \( \mu' \) with \( t|\mu \rightarrow t'|\mu' \).

**Answer:** TRUE. The extra flexibility of the new subtyping rule allows a values of some Ref type to be used in a context where another (inequivalent) Ref type is expected. But the only things that this context might do with a bogus reference value are either reading from it or storing into it. The first case causes no problems. In the second case, bad behavior may result later, when someone else reads a bad value from the store, but the assignment itself works fine.

(b) **Preservation**: If

\[
\begin{align*}
\Gamma|\Sigma & \vdash t : T \\
\Gamma|\Sigma & \vdash \mu \\
t|\mu & \rightarrow t'|\mu'
\end{align*}
\]

then, for some \( \Sigma' \supseteq \Sigma \),

\[
\begin{align*}
\Gamma|\Sigma' & \vdash t' : T \\
\Gamma|\Sigma' & \vdash \mu'.
\end{align*}
\]

**Answer:** FALSE. For example, suppose \( \Sigma = \{l_1 \mapsto \{a: \{\}\}\} \) and \( \mu = \{l_1 \mapsto \{a=\{\}\}\} \) and \( \mu' = \{l_1 \mapsto \{\}\} \). Let \( t \) be \( l_1 :={} \). Then \( \Sigma \vdash \mu \) and \( t|\mu \rightarrow \text{unit}|\mu' \), but there is no extension of \( \Sigma' \) such that \( \Sigma' \vdash \mu \).

(c) **Existence of joins**: For every pair of types \( S \) and \( T \) there is some type \( J \) such that \( S \) and \( T \) are both subtypes of \( J \) and such that, for any other type \( U \), if \( S \) and \( T \) are both subtypes of \( U \), then \( J \) is a subtype of \( U \).

**Answer:** TRUE. After this change, Ref becomes a simple covariant operator—anal ogous (in its subtyping behavior) to a single-field record. We already know that joins exist for record types, so we should expect no problems. (This is just a handwave, of course. To be completely sure, we should actually extend the proof of existence of joins [and meets]; but this was not required for this problem.)

Grading scheme:

- 5 points for each part
- -5 for incorrect answer
- 2 points for correct answer but missing or incorrect explanation
Object Encodings in Lambda-Calculus

The questions in this section are based the following small class hierarchy encoded in lambda-calculus. (Note that this encoding is in the simpler style of section 18.11 of TAPL; it does not incorporate the refinements for improved efficiency discussed at the very end of the chapter, in 18.12.)

```lambda
/* A couple of miscellaneous helper functions -- "not" on booleans... */
not = \b:Bool. if b then false else true;
/* and a comparison function for numbers: */
leq =
  fix (\f:Nat→Nat→Bool.
      \m:Nat. \n:Nat.
        if iszero m then true
        else if iszero n then false
        else f (pred m) (pred n));

/* The interface type of "pair objects": */
Pair = {set1:Nat→Unit, set2:Nat→Unit, lessoreq:Unit→Bool, greater:Unit→Bool};

/* The internal representation of "pair objects": */
PairRep = {x1: Ref Nat, x2:Ref Nat};

/* A class of "abstract pair objects." Note that the lessoreq and
greater methods call each other recursively. */
absPairClass =
  \r:PairRep.
    \self: Unit→Pair.
      \_:Unit.
        {set1 = \i:Nat. r.x1:=i,
         set2 = \i:Nat. r.x2:=i,
         lessoreq = \_:Unit. not ((self unit).greater unit),
         greater = \_:Unit. not ((self unit).lessoreq unit)};

/* A function that creates a new abstract pair object: */
newAbsPair =
  \_:Unit. let r = {x1=ref 0, x2=ref 0} in
    fix (absPairClass r) unit;

/* A subclass that overrides the lessoreq method: */
pairClass =
  \r:PairRep.
    \self: Unit→Pair.
      \_:Unit.
        let super = absPairClass r self unit in
        {set1 = super.set1,
         set2 = super.set2,
         lessoreq = \_:Unit. leq (!r.x1) (!r.x2),
         greater = super.greater};

/* A function that creates a new pair object: */
newPair =
  \_:Unit. let r = {x1=ref 0, x2=ref 0} in
    fix (pairClass r) unit;
```
6. (6 points) Circle T or F for each of the following statements.

(a) The expression `newAbsPair unit` diverges.
   T F

(b) The expression `(newAbsPair unit).set1 5` diverges.
   T F

(c) The expression `(newAbsPair unit).greater unit` diverges.
   T F

(d) The expression `newPair unit` diverges.
   T F

(e) The expression `(newPair unit).set1 5` yields `unit`.
   T F

(f) The expression `(newPair unit).greater unit` yields `false`.
   T F

7. (16 points) Write another class `myPairClass` that uses `pairClass` as its superclass and that adds one more method, called `setSmaller`, that calls the `lessoreq` method to determine which field is smaller and then calls either the `set1` or the `set2` method to update the value of this field. (Your new method should not use `:=`, `!`, or numeric comparison directly.) You do not need to write the `newMyPair` function—just the class.

```
MyPair = {set1:Nat→Unit, set2:Nat→Unit,
       lessoreq:Unit→Bool, greater:Unit→Bool,
       setSmaller:Nat→Unit};

myPairClass =
  λr:PairRep.
  λself: Unit→MyPair.
  λ_:Unit.
    let super = pairClass r self unit in
    {set1 = super.set1,
     set2 = super.set2,
     lessoreq = super.lessoreq,
     greater = super.greater,
     setSmaller =
       λn:Nat.
         if (self unit).lessoreq unit
         then (self unit).set1 n
         else (self unit).set2 n
    };
```

Grading scheme: 0-5 for getting the infrastructure right, but the method missing, or mostly wrong; 5-12 almost correct answers but wrong thanks used (e.g. `unit`) and types don’t agree; 13-16 for correct answers with small mistakes, -1 for not applying a `unit` to `lessoreq`, -1 for writing `MyPairRep` instead of `PairRep`, -1 for `Pair` instead of `MyPair`. No points off for using `super` instead of `self unit`. 

8
Featherweight Java with Exceptions

The problems in this section deal with an extension of FJ with exceptions. The definition of the original FJ is given for reference on page 6 of the companion handout.

The full syntax of terms in the extended language, including two new syntactic forms for raising and handling errors, is:

\[
\begin{align*}
t & ::= \\
x & \quad \text{variable} \\
t.f & \quad \text{field access} \\
t.m(\overline{t}) & \quad \text{method invocation} \\
\text{new } C(\overline{t}) & \quad \text{object creation} \\
(C) t & \quad \text{cast} \\
\text{error} & \quad \text{run-time error} \\
\text{try } t \text{ with } t & \quad \text{trap errors}
\end{align*}
\]

(Note that we are adding the simplest form of exceptions here—exceptions are just the term error, with no additional value carried along.)

The typing rules for error and try...with... are standard:

\[
\begin{align*}
\Gamma & \vdash \text{error} : C & \quad (\text{T-Error}) \\
\Gamma & \vdash t_1 : C, \Gamma & \vdash t_2 : C \\
\Gamma & \vdash \text{try } t_1 \text{ with } t_2 : C & \quad (\text{T-Try})
\end{align*}
\]

Having added exceptions to the system, we no longer need to define failing casts as stuck terms (as in original FJ); instead, we can make a failing cast raise an exception:

\[
\begin{align*}
\text{C } \not\ll \text{ D} & \\
(D)(\text{new } C(\overline{t})) & \rightarrow \text{error} & \quad (\text{E-BadCast})
\end{align*}
\]

The other new evaluation rules follow the same pattern as in \(\lambda\)-, with exceptions: error “percolates up” through the other term constructors, aborting their evaluation as it goes. For example:

\[
\begin{align*}
\text{error.}m(\overline{u}) & \rightarrow \text{error} & \quad (\text{E-InvkError}) \\
(\text{new } C(\overline{v})).m(u_1...u_{i-1},\text{error}, t_{i+1}...t_n) & \rightarrow \text{error} & \quad (\text{E-InvkErrorArg}) \\
\text{try error with } t_2 & \rightarrow t_2 & \quad (\text{E-TryError})
\end{align*}
\]
8. (10 points) What other evaluation rules do we need to add to complete the definition?

Answer:

\[
\begin{align*}
\text{error} & \cdot f_i \rightarrow \text{error} & \text{(E-PROJERROR)} \\
\text{new } C(v_1 \ldots v_{i-1}, \text{error}, t_{i+1} \ldots t_n) & \rightarrow \text{error} & \text{(E-NEWEERROR)} \\
(C)\text{error} & \rightarrow \text{error} & \text{(E-CASTERROR)} \\
\text{try } v_1 \text{ with } t_2 & \rightarrow v_1 & \text{(E-TRY-OK)} \\
\hline
\text{try } t_1 & \rightarrow t_1' \\
\text{try } t_1 \text{ with } t_2 & \rightarrow \text{try } t_1' \text{ with } t_2 & \text{(E-TRY-BODY)}
\end{align*}
\]

Grading scheme: 2 points per rule. (Most people missed the two last rules.)

9. (6 points) State an appropriate progress theorem for the extended language. (Do not prove it.)

Answer: Suppose \( t \) is a closed, well-typed normal form. Then either \( t \) is a value or \( t = \text{error} \).

Grading scheme: -1 for not mentioning “well-typed”, -1 for not mentioning “closed”, -2 off for missing each one of three cases: term is value, term is error, term reduces. -2 for mentioning irrelevant evaluation contexts.
10. (18 points) The statement of the preservation theorem for FJ with exceptions is exactly the same as for ordinary FJ:

**Theorem:** If $\Gamma \vdash t : C$ and $t \rightarrow t'$, then $\Gamma \vdash t' : C'$ for some $C' <: C$.

Fill in the blanks in the following proof of this theorem. Make sure to explicitly mention every step required in the proof (use of an assumption, use of the induction hypothesis, use of a typing or evaluation rule, etc.).

**Proof:** By induction on a derivation of $t \rightarrow t'$, with a case analysis on the final rule. (Just three of the cases are given here; we are eliding several others.)

**Case** E-BadCast: $t = (D)(\text{new } B(v))$  $t' = \text{error}$  $B \not<: D$

The result is immediate by T-Error and reflexivity of subtyping.

**Case** E-TryError: $t = \text{try error}$ with $t_2$  $t' = t_2$

By inspection of the typing rules, the final rule in the derivation of $\Gamma \vdash t : C$ must be T-Try, with premises $\Gamma \vdash \text{error} : C$ and $\Gamma \vdash t_2 : C$. The latter is the desired result, with a use of reflexivity of subtyping.

**Case** E-CastNew: $t = (D)(\text{new } C_0(v))$  $C_0 <: D$  $t' = \text{new } C_0(v)$

By inspection of the typing rules, it is clear that the final rule in the derivation of $\Gamma \vdash t : C$ must be T-UCast, T-DCast, or T-SCast, with $C = D$ and with subderivation $\Gamma \vdash t' : E$ (for some $E$) in each case. Again, by inspection of the typing rules, the final rule in the subderivation must be T-New, with $E = C_0$ (and with premises $\Gamma \vdash \overline{v} : \overline{C}$ and $\overline{C} <: \overline{D}$, but we do not need these). Setting $C' = C_0$ yields the desired result.

**Grading scheme:** 6 points each part, -1 for not mentioning reflexivity, -6 for random guessing, -5 lots of extraneous (wrong) stuff but some step correct, no points off for extraneous correct stuff, -4 or -3 for some steps correct but confused answers.
Polymorphism

The following problem concerns the polymorphic lambda-calculus (with a primitive fix construct and booleans). This system is summarized on page 9 of the companion handout.

11. (7 points) Suppose (following the example in Chapter 23 of TAPL and in the lecture notes) that our language is also equipped with a type constructor List and the following term constructors for the usual list manipulation primitives.

\[
\begin{align*}
nil &: \forall X. \text{List } X \\
\text{cons} &: \forall X. X \to \text{List } X \to \text{List } X \\
\text{isnil} &: \forall X. \text{List } X \to \text{Bool} \\
\text{head} &: \forall X. \text{List } X \to X \\
\text{tail} &: \forall X. \text{List } X \to \text{List } X
\end{align*}
\]

Complete the following definition of a mapfilter function on lists by filling in the missing type parameters (mapfilter is an “all in one” combination of map and filter—it filters a list using the boolean function test and applies \( f \) to each element of the resulting list). All necessary type parameters are indicated with blanks, which you are to fill in.

\[
\text{mapfilter} = \lambda X. \lambda Y. \lambda \text{test}:X \to \text{Bool}. \lambda f:X \to Y \\
(\text{fix}\ (\lambda \text{rec:List } X \to \text{List } Y. \\
\lambda \text{xs:List } X. \\
\quad \text{if isnil } [X] \text{ } \text{xs} \text{ then} \\
\quad \text{nil } [Y] \\
\quad \text{else if test } (\text{head } [X] \text{ } \text{xs}) \text{ then} \\
\quad \text{cons } [Y] (f (\text{head } [X] \text{ } \text{xs})) (\text{rec } (\text{tail } [X] \text{ } \text{xs})) \\
\quad \text{else} \\
\quad \text{rec } (\text{tail } [X] \text{ } \text{xs})))
\]

*Grading scheme: 1 point per blank*
Companion handout

Full definitions of the systems used in the exam
Untyped Lambda-calculus

Syntax
\[ t ::= \]
\[ x \]
\[ \lambda x. t \]
\[ t t \]

\[ v ::= \]
\[ \lambda x. t \]

Evaluation

\[ \frac{t_1 \rightarrow t'_1}{t_1 t_2 \rightarrow t'_1 t_2} \quad \text{(E-App1)} \]

\[ \frac{t_2 \rightarrow t'_2}{v_1 t_2 \rightarrow v_1 t'_2} \quad \text{(E-App2)} \]

\[ (\lambda x. t_1) v_2 \rightarrow [x \mapsto v_2] t_{12} \quad \text{(E-AppAbs)} \]
Simply-typed lambda calculus with subtyping
(and records and variants)

Syntax

\[ t ::= \]

\[ x \quad \text{variable} \]
\[ \lambda x:T.t \quad \text{abstraction} \]
\[ t\ t \quad \text{application} \]
\[ \{l_i=t_i\ i\in 1..n\} \quad \text{record} \]
\[ \text{unit} \quad \text{constant} \text{unit} \]
\[ \text{ref } t \quad \text{reference creation} \]
\[ !t \quad \text{dereference} \]
\[ t:=t \quad \text{assignment} \]
\[ l \quad \text{store location} \]
\[ \langle l=t\rangle\quad \text{no as} \]

\[ \text{case } t \text{ of } \langle l=x\rangle\Rightarrow t_i\ i\in 1..n \]

values

\[ v ::= \quad \text{values} \]
\[ \lambda x:T.t \quad \text{abstraction value} \]
\[ \{l_i=v_i\ i\in 1..n\} \quad \text{record value} \]
\[ \text{unit} \quad \text{constant unit} \]
\[ l \quad \text{store location} \]

Types

\[ T ::= \quad \text{types} \]
\[ \{l_i:T_i\ i\in 1..n\} \quad \text{type of records} \]
\[ \text{Top} \quad \text{maximum type} \]
\[ T\rightarrow T \quad \text{type of functions} \]
\[ \text{Unit} \quad \text{unit type} \]
\[ \text{Ref } T \quad \text{type of reference cells} \]
\[ <l_i:T_i\ i\in 1..n\> \quad \text{type of variants} \]

Type environments

\[ \Gamma ::= \quad \text{type environments} \]
\[ \emptyset \quad \text{empty type env.} \]

Stores

\[ \mu ::= \quad \text{stores} \]
\[ \emptyset \quad \text{empty store} \]
\[ \mu, l = v \quad \text{location binding} \]

Store typings

\[ \Sigma ::= \quad \text{store typings} \]
\[ \emptyset \quad \text{empty store typing} \]
\[ \Sigma, l:T \quad \text{location typing} \]

Evaluation

\[ t\mu \rightarrow t'\mu' \quad \text{E-App1} \]
\[ t_1t_2\mu \rightarrow t'_1t'_2\mu' \quad \text{E-App2} \]
\[(\lambda x : T_{11} . t_2) \mid v_2 \mapsto [x \mapsto v_2] t_{12} \mid \mu \quad (E\text{-AppAbs})\]

\[{l_i = v_i} \mid l_j \mid \mu \mapsto v_j \mid \mu \quad (E\text{-ProjRcd})\]

\[l \notin \text{dom}(\mu) \quad (E\text{-RefV})\]

\[\text{ref } v_1 \mid \mu \mapsto l[(\mu, l \mapsto v_1)] \quad (E\text{-Ref})\]

\[t_1 \mid \mu \mapsto t'_1 \mid \mu' \quad (E\text{-DerefLoc})\]

\[l : \text{ref } v_2 \mid \mu \mapsto \text{unit} \mid [l \mapsto v_2] \mu \quad (E\text{-Assign})\]

\[t_1 \mid \mu \mapsto t'_1 \mid \mu' \quad (E\text{-Deref})\]

\[l : \text{ref } t_2 \mid \mu \mapsto \text{ref } t'_2 \mid \mu' \quad (E\text{-Ref})\]

\[l : \text{ref } v_1 \mid \mu \mapsto \text{ref } t'_1 \mid \mu' \quad (E\text{-RefV})\]

\[l : \text{ref } v_2 \mid \mu \mapsto \text{ref } t'_2 \mid \mu' \quad (E\text{-RefRcd})\]

\[l : \text{ref } v_1 \mid \mu \mapsto \text{ref } t'_2 \mid \mu' \quad (E\text{-RefRcd})\]

\[\text{case } (l_j = v_j) \text{ as } T \text{ of } (l_i = x_i) \Rightarrow t_i \mapsto t_j \mid \mu \mapsto [x_j \mapsto v_j] t_j \mid \mu \quad (E\text{-CaseVariant})\]

\[\text{case } t_0 \mid \mu \mapsto t'_0 \mid \mu' \quad (E\text{-Case})\]

\[\text{case } t_0 \text{ of } (l_i = x_i) \Rightarrow t_i \mapsto t_j \mid \mu \mapsto \text{case } t'_0 \text{ of } (l_i = x_i) \Rightarrow t_i' \mapsto t'_j \mid \mu' \quad (E\text{-Case})\]

\[t_i \mid \mu \mapsto t'_i \mid \mu' \quad (E\text{-Assign})\]

\[l : \text{ref } t_1 \mid \mu \mapsto l : \text{ref } t'_1 \mid \mu' \quad (E\text{-Assign1})\]

\[l : \text{ref } v_1 \mid \mu \mapsto l : \text{ref } t'_2 \mid \mu' \quad (E\text{-Assign2})\]

\[\text{case } (l_i = v_j) \text{ as } T \text{ of } (l_i = x_i) \Rightarrow t_i \mapsto t_j \mid \mu \mapsto [x_j \mapsto v_j] t_j \mid \mu \quad (E\text{-Assign1})\]

\[\text{case } t_0 \text{ of } (l_i = x_i) \Rightarrow t_i \mapsto t_j \mid \mu \mapsto \text{case } t'_0 \text{ of } (l_i = x_i) \Rightarrow t_i' \mapsto t'_j \mid \mu' \quad (E\text{-Assign2})\]

\[\text{for each } i \quad \Gamma \vdash t_i : T_i \quad (T\text{-Rcd})\]

\[\Gamma \vdash \{ l_i = t_i \mid l_i \in \Sigma \} : \{ l_i = T_i \mid l_i \in \Sigma \} \quad (T\text{-Proj})\]

\[\Gamma \vdash t_1 : \{ l_i = T_i \mid l_i \in \Sigma \} \quad \Gamma \vdash l_j : T_j \quad (T\text{-Proj})\]

\[x : T \in \Gamma \quad (T\text{-Proj})\]

\[\Gamma \mid \Sigma \vdash x : T \quad (T\text{-Proj})\]

\[\Gamma, x : T_1 | \Sigma \vdash t_2 : T_2 \quad (T\text{-Abs})\]

\[\Gamma \mid \Sigma \vdash \lambda x : T_1 . t_2 : T_1 \rightarrow T_2 \quad (T\text{-Abs})\]

\[\Gamma | \Sigma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma | \Sigma \vdash t_2 : T_{11} \quad \Gamma | \Sigma \vdash t_1 . t_2 : T_{12} \quad (T\text{-App})\]
\[
\Gamma \vdash t : S \quad S \ll T
\]

\[
\Gamma \vdash t : T
\]  

\text{(T-Sub)}

\[
\Gamma | \Sigma \vdash \text{unit} : \text{Unit}
\]  

\text{(T-Unit)}

\[
\Sigma(l) = T_1
\]

\[
\Gamma | \Sigma \vdash l : \text{Ref } T_1
\]  

\text{(T-Loc)}

\[
\Gamma | \Sigma \vdash t_1 : T_1
\]

\[
\Gamma | \Sigma \vdash \text{ref } t_1 : \text{Ref } T_1
\]  

\text{(T-Deref)}

\[
\Gamma | \Sigma \vdash t_1 : \text{Ref } T_{11}
\]

\[
\Gamma | \Sigma \vdash t_1 : \text{Ref } T_{11}
\]

\text{(T-Deref)}

\[
\Gamma | \Sigma \vdash \text{ref } t_1 : \text{Ref } T_{11}
\]

\[
\Gamma | \Sigma \vdash t_1 : \text{Ref } T_{11}
\]

\text{(T-Deref)}

\[
\Gamma | \Sigma \vdash t_1 : \text{Ref } T_{11}
\]

\text{(T-Deref)}

\[
\Gamma | \Sigma \vdash t_1 : \text{Ref } T_{11}
\]

\text{(T-Deref)}

\[
\Gamma | \Sigma \vdash t_1 : \text{Ref } T_{11}
\]

\text{(T-Deref)}

\[
\Gamma | \Sigma \vdash t_1 : \text{Ref } T_{11}
\]

\text{(T-Deref)}

\[
\Gamma | \Sigma \vdash \text{case } t_0 \text{ of } <l_i=x_i> \Rightarrow t_i^{i \in I \ldots n} : T
\]  

\text{(T-Case)}

\text{Subtyping}

\[
S \ll T
\]  

\text{(S-RefFl)}

\[
S \ll U \quad U \ll T
\]

\[
S \ll T
\]

\text{(S-Trans)}

\[
S \ll \text{Top}
\]  

\text{(S-Top)}

\[
T_1 \ll S_1 \quad S_2 \ll T_2
\]

\[
S_1 \rightarrow S_2 \ll T_1 \rightarrow T_2
\]  

\text{(S-Arrow)}

\[
\{l_i : T_i^{i \in I \ldots n+k} \} \ll \{l_i : T_i^{i \in I \ldots n} \}
\]  

\text{(S-RecDWidth)}

\[
\text{for each } i \quad S_i \ll T_i
\]

\[
\{l_i : T_i^{i \in I \ldots n} \} \ll \{l_j : S_j^{j \in I \ldots n} \}
\]  

\text{(S-RecDDepth)}

\[
\{k_j : S_j^{j \in I \ldots n} \} \text{ is a permutation of } \{l_i : T_i^{i \in I \ldots n} \}
\]

\[
\{k_j : S_j^{j \in I \ldots n} \} \ll \{l_i : T_i^{i \in I \ldots n} \}
\]  

\text{(S-RecPerm)}

\[
T_1 \ll S_1 \quad S_1 \ll T_1
\]

\[
\text{Ref } S_1 \ll \text{Ref } T_1
\]  

\text{(S-Ref)}

\[
<l_i : T_i^{i \in I \ldots n} > \ll <l_i : T_i^{i \in I \ldots n+k}>
\]  

\text{(S-VariantWidth)}
for each $i$ \( S_i \triangleleft T_i \)
\[
\langle l_i : S_i \mid i \in 1..n \rangle \ll \langle l_i : T_i \mid i \in 1..n \rangle \quad \text{(S-VariantDepth)}
\]

\( \langle k_j : S_j \mid j \in 1..n \rangle \) is a permutation of \( \langle l_i : T_i \mid i \in 1..n \rangle \)
\[
\langle k_j : S_j \mid j \in 1..n \rangle \ll \langle l_i : T_i \mid i \in 1..n \rangle \quad \text{(S-VariantPerm)}
\]
Featherweight Java

Syntax

\[\text{CL ::= class declarations}\]

\[\text{K ::= C\{C; \ K\}\{\text{super(\text{C}); this.\text{C};}\}}\]

\[\text{M ::= C\ m(\text{C}) \{\text{return t;}}\}

\[\text{t ::= x; t.f; t.m(\text{E}); new C(\text{E}); (C) t}\]

\[\text{v ::= new C(\text{E})}\]

Subtyping

\[\begin{align*}
\text{C} < \text{C} \\
\text{C} < \text{D} & \quad \text{D} < \text{E} \\
\hline
\text{C} < \text{E}
\end{align*}\]

\[\text{CT(C) = class C extends D \{\ldots\}}\]

\[\text{C} < \text{D}\]

Field lookup

\[\text{fields(\text{Object}) = •}\]

\[\begin{align*}
\text{CT(C) = class C extends D \{\text{C}; \ K\}} \\
\text{fields(D) = D; f} \\
\hline
\text{fields(C) = D; f, C; f}
\end{align*}\]

Method type lookup

\[\begin{align*}
\text{CT(C) = class C extends D \{\text{C}; \ K\}} \\
\text{B \ m (B; x) \{\text{return t;}} \in H \\
\hline
\text{mtype(m, C) = B → B}
\end{align*}\]

\[\text{CT(C) = class C extends D \{\text{C}; \ K\}} \\
\text{m is not defined in H} \\
\hline
\text{mtype(m, C) = mtype(m, D)}
\]
Method body lookup

$$CT(C) = \text{class } C \text{ extends } D \{C \bar{x}; K \bar{M}\}$$

$$B m \langle \bar{B} \bar{x} \rangle \{\text{return } t;\} \in \bar{H}$$

$$\text{mbody}(m, C) = (x, t)$$

$$CT(C) = \text{class } C \text{ extends } D \{C \bar{x}; K \bar{M}\}$$

m is not defined in \( \bar{H} \)

$$\text{mbody}(m, C) = \text{mbody}(m, D)$$

Valid method overriding

$$\text{mtype}(m, D) = D \rightarrow D_0 \text{ implies } C = D \text{ and } C_0 = D_0$$

$$\text{override}(m, D, C \rightarrow C_0)$$

Evaluation

$$\text{fields}(C) = C \bar{f}$$

$$\langle \text{new } C(\bar{v}) \rangle . f \rightarrow i_v$$

(E-ProjNew)

$$\text{mbody}(m, C) = (x, t_0)$$

$$\langle \text{new } C(\bar{v}) \rangle . m(\bar{v}) \rightarrow [x \mapsto \bar{v}, \text{this} \mapsto \text{new } C(\bar{v})] t_0$$

(E-InvkNew)

$$C \subset D$$

$$D \langle \text{new } C(\bar{v}) \rangle \rightarrow \text{new } C(\bar{v})$$

(E-CastNew)

$$t_0 \rightarrow t'_0$$

$$t_0.f \rightarrow t'_0.f$$

(E-Field)

$$t_0 \rightarrow t'_0$$

$$t_0.m(\bar{v}) \rightarrow t'_0.m(\bar{v})$$

(E-Invk-Recv)

$$t_i \rightarrow t'_i$$

$$v_0.m(\bar{v}, t_i, \bar{v}) \rightarrow v_0.m(\bar{v}, t'_i, \bar{v})$$

(E-Invk-Arg)

$$t_i \rightarrow t'_i$$

$$\text{new } C(\bar{v}, t_i, \bar{v}) \rightarrow \text{new } C(\bar{v}, t'_i, \bar{v})$$

(E-New-Arg)

$$t_0 \rightarrow t'_0$$

$$\langle C \rangle t_0 \rightarrow \langle C \rangle t'_0$$

(E-Cast)

Term typing

$$\Gamma \vdash t : C$$

$$x : C \in \Gamma$$

$$\Gamma \vdash x : C$$

(T-Var)
\[ \Gamma \vdash t_0 : C_0 \quad \text{fields}(C_0) = C \bar{F} \]
\[ \Gamma \vdash t_0.f_i : C_i \] (T-FIELD)

\[ \Gamma \vdash t_0 : C_0 \quad \text{mtype}(m, C_0) = D \rightarrow C \]
\[ \Gamma \vdash \bar{c} : C \quad C < : D \]
\[ \Gamma \vdash t_0.m(\bar{c}) : C \] (T-INVK)

\[ \text{fields}(C) = D \bar{F} \]
\[ \Gamma \vdash \bar{c} : C \quad C < : D \]
\[ \Gamma \vdash \text{new } C(\bar{c}) : C \] (T-NEW)

\[ \Gamma \vdash t_0 : D \quad D < : C \]
\[ \Gamma \vdash (C)t_0 : C \] (T-UCAST)

\[ \Gamma \vdash t_0 : D \quad C < : D \quad C \neq D \]
\[ \Gamma \vdash (C)t_0 : C \] (T-DCAST)

\[ \Gamma \vdash t_0 : D \quad C \not< : D \quad D \not< : C \]
\[ \text{stupid warning} \]
\[ \Gamma \vdash (C)t_0 : C \] (T-SCAST)

**Method typing**

\[ \bar{x} : C, \text{this} : C \vdash t_0 : E_0 \quad E_0 < : C_0 \]
\[ CT(C) = \text{class } C \text{ extends } D \{ \ldots \} \]
\[ \text{override}(m, D, C \rightarrow C_0) \]
\[ C_0 \ m \ (C \bar{x}) \{ \text{return } t_0; \} \text{ OK in } C \]

**Class typing**

\[ K = C(\bar{D} \ g, \ C \bar{F}) \quad \{ \text{super}(\bar{g}); \text{this.}\bar{F} = \bar{F}; \} \]
\[ \text{fields}(D) = \bar{D} \ g \quad \bar{M} \text{ OK in } C \]
\[ \text{class } C \text{ extends } D \{ C \bar{F}; K \bar{M} \} \text{ OK} \]

\text{8}
Polymorphic Lambda-Calculus (with fix and booleans)

Syntax

\[ t ::= \]
\[ x \]
\[ \lambda x:T.t \]
\[ t t \]
\[ \text{let } x = t \text{ in } t \]
\[ \text{fix } t \]
\[ \text{true} \]
\[ \text{false} \]
\[ \text{if } t \text{ then } t \text{ else } t \]
\[ \lambda X.t \]
\[ t [T] \]

v ::= \[ \lambda x:T.t \]
\[ \text{true} \]
\[ \text{false} \]
\[ \lambda X.t \]

T ::= \[ \text{Bool} \]
\[ X \]
\[ T \rightarrow T \]
\[ \forall X.T \]

\[ \Gamma ::= \]
\[ \emptyset \]
\[ \Gamma, X \]

Evaluation

\[ t_1 \rightarrow t'_1 \]
\[ t_1 t_2 \rightarrow t'_1 t'_2 \] (E-App1)

\[ t_2 \rightarrow t'_2 \]
\[ v_1 t_2 \rightarrow v_1 t'_2 \] (E-App2)

\[ (\lambda x:T_1.t_{12}) v_2 \rightarrow [x \mapsto v_2]t_{12} \] (E-AppAbs)

let x = v_1 in t_2 \[ \rightarrow [x \mapsto v_1]t_2 \] (E-LetV)

fix (\lambda x:T_1.t_2)
\[ \rightarrow [x \mapsto \text{fix } (\lambda x:T_1.t_2)]t_2 \] (E-FixBeta)

\[ t_1 \rightarrow t'_1 \]
\[ \text{fix } t_1 \rightarrow \text{fix } t'_1 \] (E-Fix)

if true then t_2 else t_3 \[ \rightarrow t_2 \] (E-IfTrue)
if false then \( t_2 \) else \( t_3 \) \( \rightarrow t_3 \) \( \text{(E-IfFalse)} \)

\[ t_1 \rightarrow t'_1 \]  
if \( t_1 \) then \( t_2 \) else \( t_3 \) \( \rightarrow \) if \( t'_1 \) then \( t_2 \) else \( t_3 \) \( \text{(E-If)} \)

\( (\lambda X. t_{12}) [T_2] \rightarrow [X \mapsto T_2] t_{12} \) \( \text{(E-TApp'Tabs)} \)

**Typing**

\( x : T \in \Gamma \)  
\( \Gamma \vdash x : T \) \( \text{(T-VAR)} \)

\[ \Gamma, x : T_1 \vdash t_2 : T_2 \]  
\( \Gamma \vdash \lambda x : T_1 . t_2 : T_1 \rightarrow T_2 \) \( \text{(T-Abs)} \)

\[ \Gamma \vdash t_1 : T_1 \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \]  
\( \Gamma \vdash t_1 \rightarrow t_2 : T_{12} \) \( \text{(T-App)} \)

\[ \Gamma \vdash t_1 : T_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2 \]  
\( \Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2 \) \( \text{(T-Let)} \)

\[ \Gamma \vdash t_1 : T_1 \rightarrow T_1 \]  
\( \Gamma \vdash \text{fix } t_1 : T_1 \) \( \text{(T-Fix)} \)

\( \Gamma \vdash \text{true} : \text{Bool} \) \( \text{(T-True)} \)

\( \Gamma \vdash \text{false} : \text{Bool} \) \( \text{(T-False)} \)

\[ \Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T \]  
\( \Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T \) \( \text{(T-If)} \)

\[ \Gamma, X \vdash t_2 : T_2 \]  
\( \Gamma \vdash \lambda X. t_2 : \forall X.T_2 \) \( \text{(T-TAbs)} \)

\[ \Gamma \vdash t_1 : \forall X.T_{12} \]  
\( \Gamma \vdash t_1 [T_2] : [X \mapsto T_2] T_{12} \) \( \text{(T-TApp)} \)