Inductive Definitions

1. (4 points)
   Alfred Steeple, a promising young meta-computer scientist, is designing a new programming language based around "frobs". The set of frobs is defined by the following grammar:

   \[
   F ::= \Box \\
   \mid F \odot F \\
   \mid F \odot F \\
   \mid \circ F
   \]

   The most important property of frobs is a three-place relation called “frobnostication.” This relation, written \((F_1 \mid F_2 \mid F_3)\) and pronounced “F₁, F₂, and F₃ frobnosticate,” is the least relation closed under the following rules:

   \[
   \begin{align*}
   (\Box \mid \Box \mid \Box) & \quad (F_1 \mid F_2 \mid \circ F_3) & \quad (F_2 \mid F_1 \mid \circ F_3) \\
   (F_1 \mid F_2 \mid F_2) & \quad (F_1 \mid \circ F_2 \mid F_3) & \quad (F_1 \mid F_2 \mid \circ F_3) \\
   (F_2 \mid F_1 \mid F_2 \odot F_1) & \quad (F_1 \mid F_1 \mid \circ F_2) & \quad (F_2 \mid F_2 \mid \circ F_1) \\
   (F_1 \mid F_2 \mid F_1 \odot F_2) & \quad (F_1 \mid \circ F_1 \mid \circ F_2) & \quad (F_1 \mid \circ F_1 \mid \circ F_2)
   \end{align*}
   \]

   (Note: Do not waste time trying to understand what frobnostication “means”—Alfred is the only person who understands this.)

   For each of the following triples \(F_1, F_2,\) and \(F_3\) of frobs, write “yes” if \((F_1 \mid F_2 \mid F_3)\) is in the frobnostication relation and “no” if it is not.

   (a) \((\Box \mid \Box \mid \star \star \Box)\)  \textit{Answer: no}

   (b) \((\Box \mid \Box \mid \Box \odot \Box)\)  \textit{Answer: no}

   \textit{Grading scheme: Binary.}
2. (10 points)

Alfred’s graduate student, Igor, wrote an OCaml definition for a datatype of frobs.

```ocaml
type frob = Box
    | Circ of frob * frob
    | Diamond of frob * frob
    | Star of frob
```

Unfortunately, Igor had some trouble understanding Alfred’s inference rules when he implemented some OCaml code to mechanically check frobnostication. Cross out his mistakes and write in appropriate corrections. There is exactly one mistake in each clause.

Reminder: In OCaml, the boolean expression \( x \land y \) evaluates to \( \text{true} \) if both \( x \) and \( y \) evaluate to \( \text{true} \); the expression \( x \lor y \) evaluates to \( \text{true} \) if either \( x \) or \( y \) does.

```ocaml
let rec fnos f1 f2 f3 =
    match (f1, f2, f3) with
    | (Box, Box, Star Box)    -> false
    | (f1, Circ(f1’,f2), f3) when f1 = f1’   ->
        (fnos f1 f2 (Star f3)) \&\& (fnos f2 f1 (Star f3))
    | (f1, f2, Diamond(f1’,f2’)) when f1 = f1’ \&\& f2 = f2’   ->
        fnos f2 f1 (Star (f2,f1))
    | (f1, f1’, Star(Star f2)) when f1 = f1’   ->
        (fnos f1 f1’ (Star f2)) \| (fnos f2 f2 (Star f1))
    | (_, _, _)    -> true
```

**Answer:**

```ocaml
let rec fnos f1 f2 f3 =
    match (f1, f2, f3) with
    | (Box, Box, Star Box)    -> true
    | (f1, Circ(f1’,f2), f3) when f1 = f1’   ->
        (fnos f1 f2 (Star f3)) \| (fnos f2 f1 (Star f3))
    | (f1, f2, Diamond(f1’,f2’)) when f1 = f1’ \&\& f2 = f2’   ->
        fnos f2 f1 (Diamond (f2,f1))
    | (f1, f1’, Star(Star f2)) when f1 = f1’   ->
        (fnos f1 f1’ (Star f2)) \&\& (fnos f2 f2 (Star f1))
    | (_, _, _)    -> false
```

**Grading scheme: Binary.**
Untyped Lambda Calculus

Recall the definitions of the Church numerals and the basic operations over them from Chapter 5 of TAPL:

\[ c_0 = \lambda s. \lambda z. z; \]
\[ c_1 = \lambda s. \lambda z. s z; \]
\[ c_2 = \lambda s. \lambda z. s (s z); \]
\[ c_2 = \lambda s. \lambda z. s (s (s z)); \]
\[ \text{scc} = \lambda n. \lambda s. \lambda z. s (n s z); \]
\[ \text{iszro} = \lambda m. m (\lambda x. \text{fls}) \text{tru}; \]
\[ \text{tru} = \lambda t. \lambda f. t \]
\[ \text{fls} = \lambda t. \lambda f. f \]
\[ \text{and} = \lambda b. \lambda c. b c \text{fls}; \]
\[ \text{not} = \lambda b. b \text{fls} \text{tru} \]

Here is the Church encoding of lists from the solution to homework 4:

\[ \text{nil} = \lambda c. \lambda n. n; \]
\[ \text{cons} = \lambda h. \lambda t. \lambda c. \lambda n. c h (t c n); \]
\[ \text{head} = \lambda l. l (\lambda h. \lambda t. h) \text{fls}; \]
\[ \text{isnil} = \lambda l. l (\lambda h. \lambda t. \text{fls}) \text{tru}; \]

3. (6 points) Fill in the blanks in the following definition to yield a lambda term \text{makezeros} that, when applied to a church numeral \(c\_n\), returns a list containing the element \(c\_0\) repeated \(n\) times. For example, \text{makezeros} \(c\_3\) should behave like \text{cons} \(c\_0\) (\text{cons} \(c\_0\) (\text{cons} \(c\_0\) \text{nil})).

Your answer should use only the combinators defined above (plus applications and variables). Explicit \(\lambda\)-abstractions are not allowed.

\text{Answer:}

\[ \text{makezeros} = \lambda n. n (\text{cons } c\_0) (\text{nil}) \]

\text{Grading scheme: 4 points for } \text{cons } c\_0, 2 \text{ points for } \text{nil}. \text{Partial credits given for almost correct answers.}

4. (8 points) Fill in the blanks in the following definition to yield a lambda term \text{allzeros} that, when applied to a list of numbers, returns \text{tru} if all the numbers in the list are zeros, and otherwise returns \text{fls}. For example,

\[ \text{allzeros (cons } c\_2 (\text{cons } c\_0 (\text{cons } c\_1 \text{nil}))) \]

should evaluate to \text{fls}, and

\[ \text{allzeros (cons } c\_0 (\text{cons } c\_0 \text{nil}))) \]

should evaluate to \text{tru}.

Again, your answer should use only the combinators defined at the top of the page (plus applications and variables). Explicit \(\lambda\)-abstractions are not allowed.

\text{Answer:}\ \text{allzeros} = \lambda l. l (\lambda a. \lambda b. \text{and (iszro } a) b) \text{tru}

\text{Grading scheme: 6 points for } \text{and (iszro } a) b, 2 \text{ points for } \text{tru}. \text{Partial credits given for almost correct answers.}
Simply typed lambda-calculus

The following questions refer to the simply typed lambda-calculus (with base types \texttt{Bool} and \texttt{Nat}). The syntax, typing, and evaluation rules for this system are given on page 1 of the companion handout.

5. (9 points) Recall that the erasure of a simply typed lambda-term is the untyped term found by deleting all the type annotations on \texttt{\lambda}-abstractions.

For each of the following untyped lambda-terms, write down a simply typed term whose erasure is the given term. For example, if the given term were

\[ \lambda x. \lambda y. y \ x \]

then one possible correct answer would be:

\[ \lambda x: \texttt{Bool}. \lambda y: \texttt{Bool} \to \texttt{Bool}. \ y \ x \]

If there is no simply typed term that erases to the given untyped term, write “none.”

(a) \[ \lambda x. \lambda y. y \] \hspace{1cm} \text{Answer:} \lambda x: \texttt{Bool}. \lambda y: \texttt{Bool}. y \\
(b) \[ \lambda x. (\lambda y. y \ x) \ x \] \hspace{1cm} \text{Answer: none} \\
(c) \[ \lambda f. \lambda x. f \ x \ x \] \hspace{1cm} \text{Answer:} \lambda f: \texttt{Bool} \to \texttt{Bool} \to \texttt{Bool}. \lambda x: \texttt{Bool}. f \ x \ x

Grading scheme: Binary.
6. (8 points) A “zero” of a function \( t_1 \) is an input \( n \) such that \( t_1 \ n = 0 \). Suppose we want a new language feature that allows us to easily search for a zero of a given function. This can be done by adding a new operator \( \text{findzero} \), whose behavior can be described informally as follows:

- if \( t_1 \ n \) reduces to 0, then \( \text{findzero} \ t_1 \ n \) eventually reduces (perhaps after many steps) to \( n \);
- if \( t_1 \ n \) reduces to a numeric value that is different from 0, then \( \text{findzero} \ t_1 \ n \) reduces to \( \text{findzero} \ (\text{succ} \ n) \).

Formalize \( \text{findzero} \) by filling in the blanks in the following typing and evaluation rules. Your evaluation rules should enforce a left-to-right, call-by-value reduction strategy, and should preserve the properties of determinism and uniqueness of normal forms.

\[
\begin{align*}
\Gamma \vdash t_1 : \text{Nat} \rightarrow \text{Nat} & \quad \Gamma \vdash t_2 : \text{Nat} \\
\Gamma \vdash \text{findzero} \ t_1 \ t_2 : \text{Nat} & \quad \text{(T-FindZero)} \\
\hline
& \quad t_1 \rightarrow t'_1 \\
\hline
\text{findzero} \ t_1 \ t_2 & \rightarrow \text{findzero} \ t'_1 \ t_2 & \quad \text{(E-FindZero1)} \\
\hline
& \quad t_2 \rightarrow t'_2 \\
\hline
\text{findzero} \ v_1 \ t_2 & \rightarrow \text{findzero} \ v'_1 \ t'_2 & \quad \text{(E-FindZero2)} \\
\hline
\hline
& \quad \text{if iszero} \ (v_1 \ n_2) \ \text{then} \ n_2 \ \text{else} \ \text{findzero} \ v_1 \ (\text{succ} \ n_2) & \quad \text{(E-FindZeroV)}
\end{align*}
\]

Grading scheme: Two points for each rule.

7. (4 points) Can \( \text{findzero} \) be defined as a derived form (i.e., can we “desugar” each \( \text{findzero} \ t_1 \ t_2 \) into some ordinary term, with the same typing and evaluation behavior, in the pure simply typed lambda-calculus with booleans and numbers)? If so, give its definition. If not, briefly explain why not.

Answer:
No. If \( \ell \) has no zeros at all (e.g., \( \ell = \lambda x: \text{Nat}. \ \text{succ} \ x \)), then, according to the informal description, \( \text{findzero} \ \ell \ 0 \) must diverge. But we know that all well-typed terms in the simply typed lambda-calculus are terminating. So there is no way of translating \( \text{findzero} \) into a well-typed term of this language.

Grading scheme: -2 for the wrong answer. -1 for imperfect justification or implementation, -2 for wrong justification or implementation without recursion.
Simply typed lambda-calculus with subtyping

The following questions refer to the pure simply typed lambda-calculus with subtyping (with just Top—no records). The syntax, typing, and evaluation rules for this system are given on page 3 of the companion handout. The rules defining algorithmic the subtyping and typing relations for this system are given on page 4.

8. (21 points) Recall the inversion lemma for the subtype relation in the simply typed lambda-calculus with subtyping:

**Lemma (Subtyping Inversion):** If $S <: T_1 \rightarrow T_2$, then $S$ has the form $S_1 \rightarrow S_2$, with $T_1 <: S_1$ and $S_2 <: T_2$.

The minimal typing theorem for this system can be stated as follows:

**Theorem:** If $\Gamma \vdash t : T$, then $\Gamma \vdash ▶ t : S$ for some $S <: T$.

Give a detailed proof of the minimal typing theorem.

(Note that the system we are considering in this problem, defined on page 3 of the handout, includes just Top and $\rightarrow$; your proof need not deal with other type constructors such as records. If needed, you may refer [without proof] to the subtyping inversion lemma stated above. Also, you may assume that the declarative and algorithmic subtype relations coincide—i.e., $S <: T$ iff $\vdash ▶ S <: T$, for all $S$ and $T$).

**Answer:** By induction on the given (declarative) typing derivation, with a case analysis on the final rule used in the derivation.

---

**Case T-VAR:** $t = x$, $\Gamma(x) = T$

Immediate, by TA-VAR.

**Case T-ABS:** $t = \lambda x : T_1 . t_2$, $\Gamma, x : T_1 \vdash t_2 : T_2$, $T = T_1 \rightarrow T_2$

By the induction hypothesis, $\Gamma, x : T_1 \vdash ▶ t_2 : S_2$, for some $S_2 <: T_2$. By TA-ABS, $\Gamma \vdash ▶ t : T_1 \rightarrow S_2$. By S-ARROW, $T_1 \rightarrow S_2 <: T_1 \rightarrow T_2$, as required.

**Case T-APP:** $t = t_1 t_2$, $\Gamma \vdash t_1 : T_1 \rightarrow T_12$, $\Gamma \vdash t_2 : T_11$, $T = T_{12}$

By the induction hypothesis, $\Gamma \vdash ▶ t_1 : S_1$ for some $S_1 <: T_{11} \rightarrow T_{12}$ and $\Gamma \vdash ▶ t_2 : S_2$ for some $S_2 <: T_{11}$. By the subtyping inversion lemma, $S_1$ must have the form $S_11 \rightarrow S_12$, for some $S_{11}$ and $S_{12}$ with $T_{11} <: S_{11}$ and $S_{12} <: T_{12}$. By transitivity, $S_2 <: S_{11}$. By the completeness of algorithmic subtyping (stated above), $\vdash ▶ S_2 <: S_{11}$. Now, by TA-APP, $\Gamma \vdash ▶ t_1 t_2 : S_{12}$, which finishes this case since we already have $S_{12} <: T_{12}$.

**Case T-SUB:** $t : U$, $U <: T$

By the induction hypothesis, $\Gamma \vdash ▶ t : S$, with $S <: U$. By S-TRANS, $S <: T$.

**Grading scheme:**

- 8 for omitting T-APP case.
- 3 for omitting T-VAR case.
- 5 for omitting T-SUB case.
- 1 for not stating that the proof was by induction.
- 5 for a skeleton style proof.
- 1 to 2 for minor “thinkos”.
- 3 to 4 for serious localized confusions
- 2 for missing the use of S-TRANS in T-SUB case.
- 3 for using induction in the T-VAR case.
Simply typed lambda-calculus with subtyping, records, and references

The following questions refer to the simply typed lambda-calculus with subtyping, records, and references (and base types $\text{Nat}$, $\text{Bool}$, and $\text{Unit}$). The syntax, typing, and evaluation rules for this system are given on page 5 of the companion handout.

9. (3 points)
   (a) Is $\{x:\text{Top}\}$ a subtype of $\{x:\text{Top} \to \text{Top}\}$?  Answer: no
   (b) Is $\{x:\text{Ref Top}\}$ a subtype of $\{x:\text{Top}\}$?  Answer: yes
   (c) Is $\text{Ref Top}$ a subtype of $\text{Ref (Ref Top)}$?  Answer: no

Grading scheme: Binary.
10. (6 points) Recall the definitions of joins and meets in the subtype relation:

- A type $J$ is called a *join* of a pair of types $S$ and $T$ if $S <: J$, $T <: J$, and, for all types $U$, if $S <: U$ and $T <: U$, then $J <: U$.

- A type $M$ is a *meet* of $S$ and $T$ if $M <: S$, $M <: T$, and, for all types $L$, if $L <: S$ and $L <: T$, then $L <: M$.

Write down a join and meet for each pair of types, where they exist (otherwise write “undefined”).

(a) $S = \{ x: \{ a: \text{Nat}, b: \text{Nat} \} \}$
$T = \{ x: \{ b: \text{Top}, d: \text{Top} \} \}$

*Answer:*

- $\text{join} = \{ x: \{ b: \text{Top} \} \}$
- $\text{meet} = \{ x: \{ a: \text{Nat}, b: \text{Nat}, d: \text{Top} \} \}$

(b) $S = \{ x: \text{Ref Top} \}$
$T = \{ x: \text{Ref Nat}, y: \text{Ref Top} \}$

*Answer:*

- $\text{join} = \{ x: \text{Top} \}$
- $\text{meet} = \text{undefined}$

(c) $S = (\text{Top} \rightarrow \text{Bool}) \rightarrow (\text{Top} \rightarrow \text{Bool})$
$T = (\text{Nat} \rightarrow \text{Nat}) \rightarrow (\text{Nat} \rightarrow \text{Top})$

*Answer:*

- $\text{join} = \text{Top}$
- $\text{meet} = (\text{Nat} \rightarrow \text{Top}) \rightarrow (\text{Top} \rightarrow \text{Bool})$

*Grading scheme: Binary.*
11. (6 points) Suppose we add a new axiom

\[(\ ) \ll\text{ Top} \rightarrow\text{ Top}\]

to the rules defining the subtype relation. Does the progress theorem remain true in the new system? Briefly explain why or why not.

*Answer: Progress fails. For example, the term \((\ ) \ (\ )\) is well typed, but stuck.*

*Grading scheme: -6 for Yes. -3 for wrong or irrelevant justification.*

---

12. (6 points) Suppose, instead, that we add the subtyping axiom

\[\text{Top} \ll (\ )\]

to the original system. Does the progress theorem remain true? Briefly explain why or why not.

*Answer: Progress remains true: there is no way to do anything with a value of type \((\ )\), hence no way to test whether it is really a record (and get stuck if it is not).*

*Grading scheme: -6 for Yes. -3 for wrong or irrelevant justification.*
Featherweight Java

The following questions refer to the Featherweight Java. The syntax, typing, and evaluation rules for this system are given on page 9 of the companion handout.

Consider the following FJ class definitions (where class A is defined elsewhere).

```java
class Fst extends Object {
    A f1; A f2;
    Fst(A f1, A f2) { super(); this.f1 = f1; this.f2 = f2; }
    A choose() { return this.f1; }
    A chooseagain() { return this.choose(); }
}

class Snd extends Fst {
    Snd(A f1, A f2) { super(f1, f2); }
    A choose() { return this.f2; }
}

class Thd extends Snd {
    Thd(A f1, A f2) { super(f1, f2); }
    A choose() { return this.chooseagain(); }
}
```

13. (3 points) Suppose that v and w are values belonging to class A. Write down the final results of evaluating the following terms.

(a) (new Fst(v, w)).chooseagain() \textbf{Answer: } v
(b) (new Snd(v, w)).chooseagain() \textbf{Answer: } w
(c) (new Thd(v, w)).chooseagain() \textbf{Answer: } diverges

\textit{Grading scheme: Binary}
14. (11 points) Fill in the blanks in the following encoding of the classes Fst, Snd, and Thd into the simply-typed lambda-calculus with subtyping (and records, references, etc, as shown on page 5 of the companion handout), in the style of Section 18.11 of TAPL.

\[\text{Rep} = \{ f1: \text{Ref A, f2: Ref A} \};\]
\[\text{Fst} = \text{Snd} = \text{Thd} = \{ \text{choose: Unit} \rightarrow \text{A, chooseagain: Unit} \rightarrow \text{A} \};\]

**Answer:**

\[\text{fstClass} = \lambda r: \text{Rep.} \lambda self: \text{Unit} \rightarrow \text{Fst.} \lambda _: \text{Unit.}\]
\[\text{fstClass} = \lambda self: \text{Unit} \rightarrow \text{Fst.} \lambda _: \text{Unit.}\]
\[\text{choose} = \lambda _: \text{Unit.} !(r.f1),\]
\[\text{chooseagain} = \lambda _: \text{Unit.} (self \text{ unit}).\text{choose unit }\];

\[\text{sndClass} = \lambda r: \text{Rep.} \lambda self: \text{Unit} \rightarrow \text{Snd.} \lambda _: \text{Unit.}\]
\[\text{let super = fstClass r self unit in}\]
\[\{ \text{choose} = \lambda _: \text{Unit.} !(r.f2),\]
\[\text{chooseagain} = \text{super.\text{chooseagain }\};\]

\[\text{thdClass} = \lambda r: \text{Rep.} \lambda self: \text{Unit} \rightarrow \text{Thd.} \lambda _: \text{Unit.}\]
\[\text{let super = sndClass r self unit in}\]
\[\{ \text{choose} = \lambda _: \text{Unit.} (self \text{ unit}).\text{chooseagain unit,}\]
\[\text{chooseagain} = \text{super.\text{chooseagain }\};\]

\[\text{newFst} = \lambda f1val: \text{A.} \lambda f2val: \text{A.}\]
\[\text{let r = (f1 = ref f1val, f2 = ref f2val) in}\]
\[\text{fix (fstClass r) unit;}\]

**Grading scheme:** One point for each hole.
15. (15 points)

The substitution lemma for FJ is stated as follows:

**Lemma:** If \( \Gamma, \bar{\alpha} : E \vdash t : C \) and \( \Gamma \vdash \bar{\alpha} : \bar{\alpha} \), where \( \bar{\alpha} <: E \), then \( \Gamma \vdash [\bar{\alpha} \rightarrow \bar{\alpha}]t : E \) for some \( E <: C \).

Fill in the blanks in the following proof of this lemma.

**Proof:** By induction on the derivation of \( \Gamma, \bar{\alpha} : E \vdash t : C \).

**Case T-VAR:** \( t = x \quad x : C \in \Gamma \)

If \( x \not\in \bar{\alpha} \), then the result is trivial since \( [\bar{\alpha} \rightarrow \bar{\alpha}]x = \bar{\alpha} \). On the other hand, if \( x = x_i \) and \( C = B_{i} \), then, since \( [\bar{\alpha} \rightarrow \bar{\alpha}]x = B_i \), letting \( E = A_i \) finishes the case.

**Case T-FIELD:** \( t = t_0 \cdot f_i \quad \Gamma, \bar{\alpha} : E \vdash t_0 : C_0 \quad fields(C_0) = \bar{C}_i \quad C = C_i \)

By the induction hypothesis, there is some \( E_0 \) such that \( \Gamma \vdash [\bar{\alpha} \rightarrow \bar{\alpha}]t_0 : E_0 \) and \( E_0 <: C_0 \). It is easy to check that \( fields(E_0) = \langle fields(C_0), \bar{E}_i \rangle \) for some \( \bar{E} \). Therefore, by T-FIELD, \( \Gamma \vdash ([\bar{\alpha} \rightarrow \bar{\alpha}]t_0 \cdot f_i : C_i \).

**Case T-INVK:** \( t = t_0 \cdot m(\bar{E}) \quad \Gamma, \bar{\alpha} : E \vdash t_0 : C_0 \quad mtype(m, C_0) = \bar{C} \rightarrow C \)

By the induction hypothesis, there are some \( E_0 \) and \( E \) such that:

\[
\Gamma \vdash [\bar{\alpha} \rightarrow \bar{\alpha}]t_0 : E_0 \quad E_0 <: C_0 \quad \Gamma \vdash [\bar{\alpha} \rightarrow \bar{\alpha}]E : E \quad E <: E_0.
\]

It is easy to check that \( mtype(m, E_0) = \bar{C} \rightarrow C \). Moreover, \( E <: E \) by the transitivity of <: . Therefore, by T-INVK, \( \Gamma \vdash [\bar{\alpha} \rightarrow \bar{\alpha}]t_0 \cdot m[\bar{\alpha} \rightarrow \bar{\alpha}]E : C \).

**Case T-NEW:** \( t = new C(\bar{E}) \quad fields(C) = \bar{C}_i \quad \Gamma, \bar{\alpha} : E \vdash \bar{E} : E \quad \bar{E} <: E \)

By the induction hypothesis, \( \Gamma \vdash [\bar{\alpha} \rightarrow \bar{\alpha}]E : E \) for some \( \bar{E} \) with \( \bar{E} <: E \). We have \( E <: E \) by the transitivity of <: . Therefore, by T-NEW, \( \Gamma \vdash new C([\bar{\alpha} \rightarrow \bar{\alpha}]E) : C \).

**Case T-UCAST:** \( t = (C) \quad \Gamma, \bar{\alpha} : E \vdash t_0 : D \quad D <: C \)

By the induction hypothesis, there is some \( E \) such that \( \Gamma \vdash [\bar{\alpha} \rightarrow \bar{\alpha}]t_0 : E \) and \( E <: D \). We obtain \( E <: C \) by the transitivity of <: , which yields \( \Gamma \vdash (C) ([\bar{\alpha} \rightarrow \bar{\alpha}]t_0) : C \) by T-UCAST.

**Case T-DCAST:** \( t = (C) \quad \Gamma, \bar{\alpha} : E \vdash t_0 : D \quad D <: C \quad D \neq D \)

By the induction hypothesis, there is some \( E \) such that \( \Gamma \vdash [\bar{\alpha} \rightarrow \bar{\alpha}]t_0 : E \) and \( E <: D \). If \( E <: C \) or \( C <: E \), then \( \Gamma \vdash (C) ([\bar{\alpha} \rightarrow \bar{\alpha}]t_0) : C \) by T-UCAST or T-DCAST, respectively. On the other hand, if both \( C \neq C \) and \( C \neq E \), then \( \Gamma \vdash (C) ([\bar{\alpha} \rightarrow \bar{\alpha}]t_0) : C \) (with a stupid warning) by T-SCAST.

**Case T-SCAST:** \( t = (C) \quad \Gamma, \bar{\alpha} : E \vdash t_0 : D \quad C \neq D \quad D \neq C \)

By the induction hypothesis, there is some \( E \) such that \( \Gamma \vdash [\bar{\alpha} \rightarrow \bar{\alpha}]t_0 : E \) and \( E <: D \). It is easy to show, by induction on the subtype relation, that if \( E <: C \) and \( E \leq D \), then either \( C <: D \) or \( D \leq C \); since we know that neither of these is the case, we may conclude that \( E \neq C \). Moreover, if it were the case that \( C <: E \), then we would have (by transitivity) that \( C <: D \), which we know is not the case; so we may conclude that \( C \neq E \). Thus, \( \Gamma \vdash (C) ([\bar{\alpha} \rightarrow \bar{\alpha}]t_0) : C \) (with a stupid warning), by T-SCAST.

**Grading scheme:** Binary.
Companion handout

Full definitions of the systems used in the exam
Simply typed lambda calculus (with \texttt{Bool} and \texttt{Nat})

\textbf{Syntax}

\begin{align*}
t & ::= \\
& \quad \text{true} \\
& \quad \text{false} \\
& \quad \text{if } t \text{ then } t \text{ else } t \\
& \quad 0 \\
& \quad \text{succ } t \\
& \quad \text{pred } t \\
& \quad \text{iszero } t \\
& \quad x \\
& \quad \lambda x : T . t \\
& \quad t \ t
\end{align*}

\begin{align*}
v & ::= \\
& \quad \text{true} \\
& \quad \text{false} \\
& \quad \text{nv} \\
& \quad \lambda x : T . t
\end{align*}

\begin{align*}
\text{nv} & ::= \\
& \quad 0 \\
& \quad \text{succ } \text{nv}
\end{align*}

\begin{align*}
T & ::= \\
& \quad \text{Bool} \\
& \quad \text{Nat} \\
& \quad T \rightarrow T
\end{align*}

\begin{align*}
\Gamma & ::= \\
& \quad \emptyset \\
& \quad \Gamma, x : T
\end{align*}

\textbf{Evaluation}

\begin{align*}
\text{if true then } t_2 \text{ else } t_3 & \rightarrow t_2 \\
\text{if false then } t_2 \text{ else } t_3 & \rightarrow t_3
\end{align*}

\begin{align*}
\text{if } t_1 \text{ then } t_2 \text{ else } t_3 & \rightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3
\end{align*}

\begin{align*}
\text{succ } t_1 & \rightarrow \text{succ } t_1' \\
\text{pred } 0 & \rightarrow 0 \\
\text{pred } (\text{succ } \text{nv}_1) & \rightarrow \text{nv}_1
\end{align*}

\begin{align*}
\text{pred } t_1 & \rightarrow \text{pred } t_1'
\end{align*}
\(\text{iszero } 0 \rightarrow \text{true} \quad \text{(E-ISZEROZERO)}\)
\(\text{iszero } (\text{succ } n_1) \rightarrow \text{false} \quad \text{(E-ISZEROSUCC)}\)
\(t_1 \rightarrow t'_1\) \quad \text{(E-IsZERO)}
\(t_1 \rightarrow t'_1\)
\(t_1 \rightarrow t'_1 \quad t_2 \rightarrow t'_2\)
\(v_1 \rightarrow v'_1 \quad v_2 \rightarrow v'_2\)
\((\lambda x : T_{11} . t_{12}) \rightarrow [x \mapsto v_2] t_{12} \quad \text{(E-APPABS)}\)

**Typing**

\(\Gamma \vdash \text{true} : \text{Bool} \quad \text{(T-TRUE)}\)
\(\Gamma \vdash \text{false} : \text{Bool} \quad \text{(T-FALSE)}\)
\(\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T \quad \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T \quad \text{(T-IF)}\)
\(\Gamma \vdash 0 : \text{Nat} \quad \text{(T-ZERO)}\)
\(\Gamma \vdash t_1 : \text{Nat} \quad \Gamma \vdash \text{succ } t_1 : \text{Nat} \quad \text{(T-SUCC)}\)
\(\Gamma \vdash t_1 : \text{Nat} \quad \Gamma \vdash \text{pred } t_1 : \text{Nat} \quad \text{(T-PRED)}\)
\(\Gamma \vdash t_1 : \text{Nat} \quad \Gamma \vdash \text{iszero } t_1 : \text{Bool} \quad \text{(T-ISZERO)}\)
\(x : T \in \Gamma \quad \Gamma \vdash x : T \quad \text{(T-VAR)}\)
\(\Gamma, x : T_1 \vdash t_2 : T_2 \quad \Gamma \vdash \lambda x : T_1 . t_2 : T_1 \rightarrow T_2 \quad \text{(T-ABS)}\)
\(\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad \Gamma \vdash t_1 \rightarrow t_2 : T_{12} \quad \text{(T-APP)}\)
Pure simply typed lambda calculus with subtyping (no records)

Syntax
\[ t ::= \]
\[ x \quad \text{variable} \]
\[ \lambda x : T . t \quad \text{abstraction} \]
\[ t \ t \quad \text{application} \]
\[ \]
\[ v ::= \]
\[ \lambda x : T . t \quad \text{abstraction value} \]
\[ \]
\[ T ::= \]
\[ \text{Top} \quad \text{maximum type} \]
\[ T \rightarrow T \quad \text{type of functions} \]
\[ \]
\[ \Gamma ::= \]
\[ \emptyset \quad \text{empty context} \]
\[ \Gamma , x : T \quad \text{term variable binding} \]

Evaluation
\[ \]
\[ t \rightarrow t' \]
\[ t_1 t_2 \rightarrow t'_1 t_2 \]
\[ t_2 \rightarrow t'_2 \]
\[ v_1 t_2 \rightarrow v_1 t'_2 \]
\[ (\lambda x : T_11 . t_{12}) v_2 \rightarrow [x \mapsto v_2] t_{12} \]

Subtyping
\[ S < : S \]
\[ S < : U \quad U < : T \]
\[ S < : T \]
\[ S < : \text{Top} \]
\[ T_1 < : S_1 \quad S_2 < : T_2 \]
\[ S_1 \rightarrow S_2 < : T_1 \rightarrow T_2 \]

Typing
\[ x : T \in \Gamma \]
\[ \Gamma \vdash x : T \]
\[ \Gamma , x : T_1 \vdash t_2 : T_2 \]
\[ \Gamma \vdash \lambda x : T_1 . t_2 : T_1 \rightarrow T_2 \]
\[ \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \]
\[ \Gamma \vdash t_1 t_2 : T_{12} \]
\[ \Gamma \vdash t : S \quad S < : T \]
\[ \Gamma \vdash t : T \]

Terms
\[ \text{variable} \]
\[ \text{abstraction} \]
\[ \text{application} \]

Values
\[ \text{abstraction value} \]

Types
\[ \text{maximum type} \]
\[ \text{type of functions} \]

Contexts
\[ \text{empty context} \]
\[ \text{term variable binding} \]
Pure simply typed lambda calculus with subtyping (no records) — algorithmic rules

Algorithmic subtyping

\[ \begin{align*}
\Gamma \vdash S <: T \\
\Gamma \vdash T_1 <: S_1 & \quad \vdash S_2 <: T_2 \\
\Gamma \vdash S_1 \rightarrow S_2 <: T_1 \rightarrow T_2
\end{align*} \]

Algorithmic typing

\[ \begin{align*}
\Gamma \vdash t : T \\
x : T \in \Gamma & \quad \vdash x : T \\
\Gamma, x : T_1 \vdash t_2 : T_2 & \quad \Gamma \vdash \lambda x : T_1 . t_2 : T_1 \rightarrow T_2 \\
\Gamma \vdash t_1 : T_1 & \quad T_1 = T_{11} \rightarrow T_{12} \\
\Gamma \vdash t_2 : T_2 & \quad \vdash t_2 <: T_{11} \\
\Gamma \vdash t_1 \, t_2 : T_{12}
\end{align*} \]
Simply typed lambda calculus with subtyping
(and records, references, recursion, booleans, numbers)

Syntax
\[ t ::= \]
\[ x \]
\[ \lambda x:T.t \]
\[ t \ hk \]
\[ \{l_i = t_i | i \in \ldots n\} \]
\[ t.\ell \]
\[ \text{unit} \]
\[ \text{ref } t \]
\[ !t \]
\[ t:=t \]
\[ l \]
\[ \text{true} \]
\[ \text{false} \]
\[ \text{if } t \text{ then } t \text{ else } t \]
\[ 0 \]
\[ \text{succ } t \]
\[ \text{pred } t \]
\[ \text{iszero } t \]
\[ \text{let } x=t \text{ in } t \]
\[ \text{fix } t \]

\[ v ::= \]
\[ \lambda x:T.t \]
\[ \{l_i = v_i | i \in \ldots n\} \]
\[ \text{unit} \]
\[ l \]
\[ \text{true} \]
\[ \text{false} \]
\[ \text{nv} \]

\[ T ::= \]
\[ \{l_i:T_i | i \in \ldots n\} \]
\[ \text{Top} \]
\[ T \rightarrow T \]
\[ \text{Unit} \]
\[ \text{Ref } T \]
\[ \text{Bool} \]
\[ \text{Nat} \]

\[ \Gamma ::= \]
\[ \emptyset \]
\[ \Gamma, x:T \]

\[ \mu ::= \]
\[ \emptyset \]
µ, l = v

Σ ::= 0
Σ, l : T

nv ::= 0
successor nv

Evaluation

\[ \frac{t_1 | \mu \rightarrow t_1' | \mu'}{t_1 \ t_2 | \mu \rightarrow t_1' \ t_2' | \mu'} \]  \hspace{1cm} (E-APP1)

\[ \frac{t_2 | \mu \rightarrow t_2' | \mu'}{v_1 \ t_2 | \mu \rightarrow v_1 \ t_2' | \mu'} \]  \hspace{1cm} (E-APP2)

\[ \frac{(\lambda x : T \ t_1 \ t_2 | \mu) \rightarrow [x \mapsto v_2] t_1 | \mu}{(l_1 = v_1 \ {^{\text{ref} \cdot \text{ref}}}) \ . \ l_1 | \mu \rightarrow v_3 | \mu} \]  \hspace{1cm} (E-APPABS)

\[ \frac{t_1 | \mu \rightarrow t_1' | \mu'}{t_1 \ . l | \mu \rightarrow t_1' \ . l | \mu'} \]  \hspace{1cm} (E-PROJ)

\[ \frac{l \notin \text{dom}(|\mu|)}{\text{ref} \ v_1 | \mu \rightarrow l | \mu} \]  \hspace{1cm} (E-REFV)

\[ \frac{t_1 | \mu \rightarrow t_1' | \mu'}{\text{ref} \ t_1 | \mu \rightarrow \text{ref} \ t_1' | \mu'} \]  \hspace{1cm} (E-REF)

\[ \frac{|\mu(l)| = v}{!l | \mu \rightarrow v | \mu} \]  \hspace{1cm} (E-DEREFLOC)

\[ \frac{t_1 | \mu \rightarrow t_1' | \mu'}{!t_1 | \mu \rightarrow !t_1' | \mu'} \]  \hspace{1cm} (E-DEREF)

\[ \frac{l : = v_2 | \mu \rightarrow \text{unit} | [l \mapsto v_2] \mu}{t_1 | \mu \rightarrow t_1' | \mu'} \]  \hspace{1cm} (E-ASSIGN)

\[ \frac{t_1 : = t_2 | \mu \rightarrow t_1' : = t_2' | \mu'}{t_2 | \mu \rightarrow t_2' | \mu'} \]  \hspace{1cm} (E-ASSIGN1)

\[ \frac{v_1 : = t_2 | \mu \rightarrow v_1' : = t_2' | \mu'}{t_2 | \mu \rightarrow t_2' | \mu'} \]  \hspace{1cm} (E-ASSIGN2)

\[ \frac{\text{if true then } t_2 \ \text{else } t_3 | \mu \rightarrow t_2 | \mu}{\text{if false then } t_2 \ \text{else } t_3 | \mu \rightarrow t_3 | \mu} \]  \hspace{1cm} (E-IFTRUE)

\[ \frac{\text{if } t_1 \ \text{then } t_2 \ \text{else } t_3 | \mu \rightarrow \text{if } t_1' \ \text{then } t_2 \ \text{else } t_3 | \mu'}{\text{if } t_1 \ \text{then } t_2 \ \text{else } t_3 | \mu \rightarrow \text{if } t_1' \ \text{then } t_2 \ \text{else } t_3 | \mu'} \]  \hspace{1cm} (E-IF)
\[\begin{align*}
\text{(E-Succ)} & \quad t_1|\mu \rightarrow t'_1|\mu' \\
\text{(E-PredZero)} & \quad \text{pred } 0|\mu \rightarrow 0|\mu \\
\text{(E-PredSucC)} & \quad \text{pred } (\text{succ } n_1)|\mu \rightarrow n_1|\mu \\
\text{(E-Pred)} & \quad \text{pred } t_1|\mu \rightarrow \text{pred } t'_1|\mu' \\
\text{(E-IsZeroZero)} & \quad \text{iszero } 0|\mu \rightarrow \text{true}|\mu \\
\text{(E-IsZeroSucC)} & \quad \text{iszero } (\text{succ } n_1)|\mu \rightarrow \text{false}|\mu \\
\text{(E-IsZero)} & \quad \text{iszero } t_1|\mu \rightarrow \text{iszero } t'_1|\mu' \\
\text{(E-Let)} & \quad \text{let } x=v_1 \text{ in } t_2|\mu \rightarrow [x \mapsto v_1]t_2|\mu \\
\text{(E-LetV)} & \quad \text{let } x=t_1 \text{ in } t_2|\mu \rightarrow \text{let } x=t'_1 \text{ in } t_2|\mu' \\
\text{(E-FixBeta)} & \quad \text{fix } (\lambda x:T_1.t_2)|\mu \\
\text{(E-Fix)} & \quad \text{fix } t_1|\mu \rightarrow \text{fix } t'_1|\mu' \\
\text{(S-Refl)} & \quad S <: S \\
\text{(S-Trans)} & \quad S <: U \quad U <: T \\
\text{(S-Top)} & \quad S <: \text{Top} \\
\text{(S-Arrow)} & \quad T_1 <: S_1 \quad S_2 <: T_2 \\
\text{(S-RcdWidth)} & \quad S_1 \rightarrow S_2 <: T_1 \rightarrow T_2 \\
\text{(S-RcdDepth)} & \quad \{l_i:T_i\}_{i \in 1..n+k} <: \{l_i:T_i\}_{i \in 1..n} \quad \text{for each } i_S i <: T_i \\
\text{(S-RcdPerm)} & \quad \{k_j:S_j\}_{j \in 1..n} \text{ is a permutation of } \{l_i:T_i\}_{i \in 1..n} \\
\end{align*}\]
for each $i \quad \Gamma \vdash t_i : T_i$
\[ \begin{align*} 
\Gamma \vdash \{l_i = t_i\} : \{l_i : T_i\} \\
\Gamma \vdash t_1 : \{l_i : T_i\} \\
x : T \in \Gamma \\
\Gamma \vdash x : T \\
\Gamma, x : T_1 \vdash t_2 : T_2 \\
\Gamma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2 \\
\Gamma \vdash t_1 : T_1 \rightarrow T_{12} \\
\Gamma \vdash t_2 : T_{12} \\
\Gamma \vdash t : S \\
\Gamma \vdash \text{unit} : \text{Unit} \\
\Gamma \vdash \text{true} : \text{Bool} \\
\Gamma \vdash \text{false} : \text{Bool} \\
\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T \\
\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2 \\
\Gamma \vdash \text{fix } t_1 : T_1 \rightarrow T_1 
\end{align*} \]
Featherweight Java

Syntax

class declarations

CL ::= class C extends C { C(T); K M }

constructor declarations

K ::= C(T) { super(T); this.T=T; }

method declarations

M ::= C m(C X) { return t; }

terms

\[ t ::= x \]
\[ t. f \]
\[ t.m(T) \]
\[ new C(T) \]
\[ (C) t \]

values

\[ v ::= new C(V) \]

Subtyping

\[ C <: C \]
\[ C <: D \quad D <: E \]
\[ C <: E \]

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

Field lookup

\[ \text{fields} (\text{object}) = \bullet \]
\[ CT(C) = \text{class C extends D } \{ C(T); K M \} \]
\[ \text{fields} (D) = D(T) \]
\[ \text{fields} (C) = D(T), C(T) \]

Method type lookup

\[ CT(C) = \text{class C extends D } \{ C(T); K M \} \]
\[ B m(B X) \{ \text{return } t; \} \in M \]
\[ \text{mtype} (m, C) = B \rightarrow B \]
\[ CT(C) = \text{class C extends D } \{ C(T); K M \} \]
\[ m \text{ is not defined in } M \]
\[ \text{mtype} (m, C) = \text{mtype} (m, D) \]

Method body lookup

\[ \text{mbody} (m, C) = [X, t] \]
\[
CT(C) = \text{class } C \text{ extends } D \{ \overline{C} \overline{F}; K \overline{M} \}
\]
\[
B \text{ m } \{ \text{return } t; \} \in \overline{M}
\]
\[
mbody(m, C) = (\overline{x}, t)
\]
\[
CT(C) = \text{class } C \text{ extends } D \{ \overline{C} \overline{F}; K \overline{M} \}
\]
\[
m \text{ is not defined in } \overline{M}
\]
\[
mbody(m, C) = mbody(m, D)
\]

Valid method overriding
\[
mtype(m, D) = \overline{D} \rightarrow \overline{D}_0 \text{ implies } \overline{C} = \overline{D} \text{ and } C_0 = D_0
\]
\[
over\overline{D}(m, D, \overline{C} \rightarrow C_0)
\]

Evaluation
\[
fields(C) = \overline{C} \overline{F}
\]
\[
(new C(\overline{V})) \cdot f_i \rightarrow v_i
\]
\[
mbody(m, C) = (\overline{x}, t_0)
\]
\[
(new C(\overline{V})) \cdot m(\overline{U}) \rightarrow [\overline{x} \mapsto \overline{U}, \text{this} \mapsto \text{new } C(\overline{V})] t_0
\]
\[
C <: D
\]
\[
(new C(\overline{V})) \rightarrow \text{new } C(\overline{V})
\]
\[
t_0 \rightarrow t'_0
\]
\[
t_0.f \rightarrow t'_0.f
\]
\[
t_0 \rightarrow t'_0
\]
\[
t_0.m(\overline{V}) \rightarrow t'_0.m(\overline{V})
\]
\[
t_i \rightarrow t'_i
\]
\[
v_0.m(\overline{V}, t_i, \overline{E}) \rightarrow v_0.m(\overline{V}, t'_i, \overline{E})
\]
\[
t_i \rightarrow t'_i
\]
\[
\text{new } C(\overline{V}, t_i, \overline{E}) \rightarrow \text{new } C(\overline{V}, t'_i, \overline{E})
\]
\[
t_0 \rightarrow t'_0
\]
\[
(C) t_0 \rightarrow (C) t'_0
\]

Term typing
\[
\Gamma \vdash t : C
\]
\[
x : C \in \Gamma
\]
\[
\Gamma \vdash x : C
\]
\[
\Gamma \vdash t_0 : C_0 \text{ fields}(C_0) = \overline{C} \overline{F}
\]
\[
\Gamma \vdash t_0.f_i : C_i
\]
\[
\Gamma \vdash t_0 : C_0
\]
\[
mtype(m, C_0) = \overline{D} \rightarrow C
\]
\[
\Gamma \vdash C : \overline{C} \quad \overline{C} <: \overline{D}
\]
\[
\Gamma \vdash t_0.m(\overline{E}) : C
\]
\[
\begin{align*}
\text{fields}(C) &= \bar{D} \\
\Gamma \vdash \tau : \bar{C} & \quad \bar{C} <: \bar{D} \\
\Gamma \vdash \text{new } C(\bar{C}) : C & \quad \text{(T-NEW)} \\
\Gamma \vdash t_0 : D & \quad D <: \bar{C} \\
\Gamma \vdash (C)t_0 : C & \quad \text{(T-UCAST)} \\
\Gamma \vdash t_0 : D & \quad C <: D & \quad C \neq D \\
\Gamma \vdash (C)t_0 : C & \quad \text{(T-DCAST)} \\
\Gamma \vdash t_0 : D & \quad C \neq D & \quad D \neq C \\
\Gamma \vdash (C)t_0 : C & \quad \text{stupid warning} & \quad \text{(T-SCAST)} \\
\end{align*}
\]

\textit{Method typing}

\[
\begin{align*}
\text{OK in } C \\
\text{x:}\bar{C}, \text{this:}\bar{C} \vdash t_0 : E_0 & \quad E_0 <: C_0 \\
CT(C) &= \text{class } C \text{ extends } D \{\ldots\} \\
\text{override}(m, D, \bar{C}\rightarrow\bar{C}_0) & \\
C_0 \text{ m (}\bar{C}\bar{x}) \{\text{return } t_0;\} & \text{OK in } C \\
\end{align*}
\]

\textit{Class typing}

\[
\begin{align*}
\text{OK in } C \\
K &= C(\bar{D}, \bar{C}, \bar{T}) \quad \{\text{super (}\bar{D}); \text{this.}\bar{T} = \bar{T};\} \\
\text{fields}(D) &= \bar{D} \bar{y} & \text{OK in } C \\
\text{class } C \text{ extends } D \{\bar{C}\bar{T}; K\bar{M}\} & \text{OK}
\end{align*}
\]