

# CIS 500 — Software Foundations

## Final Exam (and WPE-I exam)

November 13, 2002

If you are taking this exam for WPE-I credit...

WPE ID number: \_\_\_\_\_

Otherwise...

Name: \_\_\_\_\_

Email \_\_\_\_\_

	Score
1	
2	
3	
4	
5	
6	
7	
8	
9	
Total	

## Instructions

- This is a closed-book exam: you may not make use of any books or notes.
- 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 may be used as a scratch pad.
- Some of the problems are split across two pages, with the first page providing definitions or explanations that may be needed in answering the questions on the second page. You may find it helpful to tear the first page of these problems out of the exam so that you can use it for reference.  
If you do this, please be careful to *tear* the page rather than *pulling* it out (thus destroying the staple, which will cause your exam to come loose in a box with 80 others...).
- Good luck!!

1. (20 points) Suppose we define a set of expressions called *widgets* as follows. (We'll use the metavariables  $X, Y, A, B,$  and  $C$  to range over widgets.)

$$\begin{aligned}
 X & ::= X \otimes X \\
 & \quad X \square X \\
 & \quad *
 \end{aligned}$$

We can express the syntax of widgets equivalently in OCaml, as follows:

```

type widget = Cross of widget * widget
            | Box of widget * widget
            | Star

```

The following OCaml function computes a relation on widgets, written  $X \rightarrow Y$  in mathematical notation and `lolli widX widY` in ascii. (Note: don't waste time figuring out what the  $\rightarrow$  relation "means": it was invented just for the exam and is not intended to correspond to anything computationally natural.)

```

let rec lolli widX widY =
  widX = widY ||
  match (widX, widY) with
  | (Cross(widA, widB), Cross(widC, widD)) →
    (lolli widA widC) || (lolli widB widD)
  | (Box(widA, Star), Star) →
    lolli widA Star
  | (Box(widA, Cross(widB, widC)), _) →
    lolli (Cross(widA, widY)) (Box(widC, widB))
  | (_, Box(widB, widC)) →
    (lolli widB Star) && (lolli widC widX)
  | (_,_) →
    false

```

Recall that the OCaml operator `&&` is "boolean and" and `||` is "or."

*continued...*

WPE id/last name: Give a set of inference rules for the  $\rightarrow$  relation corresponding to the given OCaml implementation—i.e., such that  $X \rightarrow Y$  iff evaluating `lolli widX widY` in OCaml yields `true` (where `widX` is an OCaml value of type `widget` representing the mathematical widget  $X$  and similarly `widY` represents  $Y$ ). One of the required rules is given as an example; you write the rest.

$$\frac{A \rightarrow \star}{A \square \star \rightarrow \star}$$

Grading scheme:

- 0 points for omitted rule
- 4 points for each rule with the following deductions:
  - -2 for each upside-down rule
  - -2 for wild card or blank where meta variable should be
  - -1 for incorrect variable other than wild card
- No deductions were made for the following
  - full credit (8/8) for folding to "or" rules together into 1 rule with an "or" separating the premises
  - attempting a rule for the last case in the OCaml match
  - No deduction for minor, but clear, notational deviations, including a different arrow character or a blank, or ordered pair notation instead of lollipop arrow
  - no deduction for omitting parens from 4th rule in key (a surprisingly rare error!)

2. Recall that the *Church encoding* of the natural numbers in untyped lambda-calculus represents a number  $n$  as a function that applies a given “successor operation”  $n$  times to a given “starting value.”

Church numerals:

$$\begin{aligned} 0 &= \lambda s. \lambda z. z \\ 1 &= \lambda s. \lambda z. (s z) \\ 2 &= \lambda s. \lambda z. s (s z) \\ 3 &= \lambda s. \lambda z. s (s (s z)) \end{aligned}$$

In this problem, we explore a different encoding of natural numbers based on their *binary* representation. Example encodings in this new system are:

Binary numerals:

$$\begin{aligned} 0 &= \lambda one. \lambda zero. \lambda t. zero t \\ 1 &= \lambda one. \lambda zero. \lambda t. one t \\ 2 &= \lambda one. \lambda zero. \lambda t. one (zero t) \\ 3 &= \lambda one. \lambda zero. \lambda t. one (one t) \\ 4 &= \lambda one. \lambda zero. \lambda t. one (zero (zero t)) \\ 5 &= \lambda one. \lambda zero. \lambda t. one (zero (one t)) \end{aligned}$$

As in the standard Church encoding, numbers are represented as functions that can be used to “count.” However, rather than applying a single given function  $n$  times to a starting value, the binary encoding is supplied with *two* functions plus a starting value. It applies its one argument to the starting value if the lowest-order bit of the binary representation of  $n$  is 1; otherwise (if the lowest order bit is 0) it applies its *zero* argument. To the result, it applies *one* if the next lowest bit in the binary representation of  $n$  is 1, otherwise *zero*; and so on up to the highest-order bit.

Note that, leading zeros are considered insignificant in this encoding—i.e., both

$$\text{fn one. fn zero. fn t. zero (one t)}$$

and

$$\text{fn one. fn zero. fn t. one t}$$

are valid encodings of the number 1. (This implies that the function passed for the *zero* argument should “have no effect” if it is the last one to be applied.)

*continued...*

WPE id / ~~last name~~ (a) (7 points) Write a function that multiplies a number in the binary encoding by 2. (Hint: in base 2, a multiplication by 2 is just a one-bit shift.)

*Grading scheme:*

- 2 points for correct args./shape of the function
- 3 points for using the term *(zero t)* in roughly the correct way
- 2 points for having everything else right

(b) (7 points) Recall that we can encode Boolean values in the untyped lambda calculus as follows:

```
tru = λt. λf. t
fls = λt. λf. f
```

Write a function that returns `tru` when given the binary encoding of 0 and `fls` if given the encoding of any non-zero number.

*Grading scheme:*

- 2 points for correct args./shape of the function
- 1 point for replacing *'one'* by something like " $\lambda x. fls$ "
- 1 point for replacing *'zero'* by something like " $\lambda x. x$ "
- 1 point for replacing *'t'* by something like "`tru`"
- 2 points for everything else right extra points for being close

The definition of the simply typed lambda-calculus is reproduced on pages 18 to 20.

3. (4 points)

For each of the following types, write a closed lambda-term that has that type *and that contains at least one application*. For example, for the type  $(\text{Nat} \rightarrow \text{Nat}) \rightarrow \text{Nat}$ , the following is a correct answer:

$\lambda f:\text{Nat} \rightarrow \text{Nat}. f\ 1$

On the other hand,

$\lambda f:\text{Nat} \rightarrow \text{Nat}. 1$

is not an acceptable answer (even though it has the correct type) because it does not contain an application.

(a)  $((\text{Nat} \rightarrow \text{Unit}) \rightarrow \text{Nat}) \rightarrow \text{Nat}$

(b)  $(\text{Unit} \rightarrow \text{Nat}) \rightarrow ((\text{Unit} \rightarrow \text{Nat}) \rightarrow \text{Nat}) \rightarrow \text{Nat}$

*Grading scheme: Two points for each part.*

4. (20 points) In this question, you will be asked to fill in a proof of the preservation theorem for the simply typed lambda calculus with records and subtyping. The theorem is identical to the one in Chapter 15 of TAPL, but for the sake of variety we'll do proof by induction on evaluation derivations (in TAPL it went by induction on typing derivations).

The following two lemmas will be used in the proof. (Note that the first four cases of the first lemma list inversion properties of the typing relation, while the last case concerns the subtype relation.)

LEMMA [INVERSION]:

- (a) If  $\Gamma \vdash \{k_i = s_i^{i \in 1..m}\} : T$ , then there are types  $T_1$  to  $T_m$  such that  $\Gamma \vdash s_i : T_i$  for each  $i$  and such that  $\{k_i : T_i^{i \in 1..m}\} <: T$ .
- (b) If  $\Gamma \vdash \tau. \lambda : T$ , then  $\Gamma \vdash \tau : \{\lambda : T\}$ .
- (c) If  $\Gamma \vdash \tau_1 \tau_2 : T$  then  $\Gamma \vdash \tau_1 : T' \rightarrow T$  and  $\Gamma \vdash \tau_2 : T'$ .
- (d) If  $\Gamma \vdash \lambda x : S_1. s_2 : T_1 \rightarrow T_2$ , then  $T_1 <: S_1$  and  $\Gamma, x : S_1 \vdash s_2 : T_2$ .
- (e) If  $S <: \{\lambda_j : T_j^{j \in 1..m}\}$ , then  $S = \{k_i : S_i^{i \in 1..n}\}$ , where  $\{\lambda_j^{j \in 1..m}\} \subseteq \{k_i^{i \in 1..n}\}$  and  $S_i <: T_j$  for each common label  $k_i = \lambda_j$ .

LEMMA [SUBSTITUTION]: If  $\Gamma, x : S \vdash \tau : T$  and  $\Gamma \vdash s : S$ , then  $\Gamma \vdash [x \mapsto s]\tau : T$ .

For each case in the proof on the next page, write down the *skeleton* of the argument. A skeleton contains the same sequence of steps as the full argument, but omits all details. The rules for writing skeletons are as follows:

- Steps of the form “By part (x) of the inversion lemma, we obtain...” in the full argument become “inversion(x)” in the skeleton.
- Steps of the form “By the substitution lemma, we obtain...” become “substitution.”
- Steps of the form “By the induction hypothesis, we obtain...” become “IH.”
- Steps of the form “By typing rule T-XXX, we obtain...” become “T-XXX.”
- If the full argument doesn't use any of the lemmas or the induction hypothesis, then its skeleton is “Direct.”

For example, if the full argument is

$$\text{Case E-APP1: } \tau = \tau_1 \tau_2 \quad \tau' = \tau'_1 \tau_2 \quad \tau_1 \rightarrow \tau'_1$$

By part (c) of the inversion lemma, we obtain  $\Gamma \vdash \tau_1 : T' \rightarrow T$  and  $\Gamma \vdash \tau_2 : T'$ . By the induction hypothesis, we obtain  $\Gamma \vdash \tau'_1 : T' \rightarrow T$ . The required result ( $\Gamma \vdash \tau'_1 \tau_2 : T$ ) now follows by T-APP.

the skeleton is written:

$$\text{Case E-APP1: } \tau = \tau_1 \tau_2 \quad \tau' = \tau'_1 \tau_2 \quad \tau_1 \rightarrow \tau'_1$$

Inversion(c), IH, T-APP.

*continued...*



8

THEOREM [PRESERVATION]: If  $t \rightarrow t'$  and  $\Gamma \vdash t : T$ , then  $\Gamma \vdash t' : T$ . *WPE id / last name: \_\_\_\_\_*

*Proof:* By induction on evaluation derivations, with a case analysis on the final rule used.

Case E-APP1:  $t = t_1 t_2 \quad t' = t'_1 t_2 \quad t_1 \rightarrow t'_1$

Inversion(c), IH, T-APP.

Case E-APP2:  $t = v_1 t_2 \quad t' = v_1 t'_2 \quad t_2 \rightarrow t'_2$

Case E-APPABS:  $t = (\lambda x : T_{11}. t_{12}) v_2 \quad t' = [x \mapsto v_2] t_{12}$

Case E-RCD:  $t = \{\lambda_i = v_i^{i \in 1..j-1}, \lambda_j = t_j, \lambda_k = t_k^{k \in j+1..n}\}$   
 $t' = \{\lambda_i = v_i^{i \in 1..j-1}, \lambda_j = t'_j, \lambda_k = t_k^{k \in j+1..n}\}$   
 $t_j \rightarrow t'_j$

Case E-PROJ:  $t = t_1 . \lambda \quad t' = t'_1 . \lambda \quad t_1 \rightarrow t'_1$

Case E-PROJRCD:  $t = \{\lambda_i = v_i^{i \in 1..n}\} . \lambda_j \quad t' = v_j$

*Grading scheme: 4 points per case. -2 for missing or extra IH. -1 for missing rule. -1 for extraneous rule.*

WPE id / last name: ~~Exceptions. Instead of counting off 2 points for a wrong but similar rule (i.e. the right rule is missing and the wrong rule extraneous) only deducted 1 point. Also, only 1 point deducted for rules in transposed order.~~

105. (3 points) Suppose we changed part (a) of the inversion lemma in ~~Problem 4~~ <sup>WPE id / last name</sup> to match the one given in chapter 15 of TAPL:

If  $\Gamma \vdash \{k_a = s_a \mid a \in 1..m\} : \{\tau_i : T_i \mid i \in 1..n\}$ , then  $\{\tau_i \mid i \in 1..n\} \subseteq \{k_a \mid a \in 1..m\}$  and  $\Gamma \vdash s_a : T_i$  for each common label  $k_a = \tau_i$ .

Which step would *fail* in the above proof of preservation? Briefly explain why.

*Grading scheme: 1 point for mentioning E-RCD; 0 points if E-RCD not mentioned. -1 for mentioning cases besides E-RCD. One additional point for an incomplete but not wrong explanation. Full credit for explaining any of the difficulties listed in the sample answer above. (Strictly speaking, only the first difficulty is correct: the new inversion lemma simply cannot be applied. But we still gave full credit to answers that focused on the other difficulties.)*

6. (3 points) Another natural-looking variant of part (a) of the inversion lemma is this:

If  $\Gamma \vdash \{k_i = s_i \mid i \in 1..n\} : T$ , then  $T = \{k_i : T_i \mid i \in 1..n\}$  and  $\Gamma \vdash s_i : T_i$  for each  $i \in 1..n$ .

Sadly, however, this variant is false.

Briefly explain why.

*Grading scheme: This question was a bit hard to answer because the clear-cut part (the fact that the lemma is problematic because of being false) was already given. Full credit was awarded for any answer that suggested that the possibility of subsumption was not taken into account, or that gave a correct counter-example. 2 points for "sort of correct" but garbled answers. 1 point for very brief or garbled answers that mention the word "subtyping." No points for explaining that the proof of preservation doesn't go through (it does go through, if we assume the lemma is true).*

7. Suppose we add new types A, B, C, D, and E to the simply typed lambda-calculus with subtyping, along with the following subtyping rules:

$$A <: B$$

$$B <: C$$

$$D <: E$$

$$A <: E$$

- (a) (6 points) For each type S from the left-hand column and type T from the right-hand column such that  $S <: T$ , draw a line connecting S to T.

Choices for S:

 $\{x:D\} \rightarrow \{y:A\}$ 
 $\{y:B\}$ 
 $\{x:A, y:A\}$ 
 $D$ 
 $\{x:B\}$ 

Choices for T:

 $E$ 
 $\{x:D\}$ 
 $\{x:E\} \rightarrow \{y:B\}$ 
 $\{\}$ 
 $\{y:E\}$ 

*Grading scheme: One point off for each missing line; one point off for each incorrect line.*

*continued...*

(b) (6 points) Recall the definitions of joins and meets in the subtype relation: ~~WPE id / last name: \_\_\_\_\_~~

- 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  $L <: M$ .

Give the join and meet of each pair of types, where they exist (otherwise write “undefined.”)

i.  $\{x:E, y:C\}$  and  $\{x:B, y:B\}$

join =

meet =

ii.  $\{x:B\}$  and  $\{y:B\}$

join =

meet =

iii.  $\{x:A\}$  and  $\{x:D\}$

join =

meet =

*Grading scheme: 1 point for each exactly correct answer. No partial credit.*

(c) (3 points) How does the join of  $\{x:A\}$  and  $\{x:D\}$  change if we add the rule  $D <: B$  to the subtyping judgment?

*Grading scheme:*

- full credit for “becomes undefined”
- no points for giving a join
- 1 point for saying something like “there are 2 possible joins”
- 1/2 points deducted for saying “undefined” plus something incorrect
- full credit for “unchanged” on papers where 7Biii was (incorrectly) marked as undefined
- 1 point for  $\{ \}$  with a semi-reasonable explanation

This problem involves an encoding of simple objects in the simply typed lambda-calculus with subtyping, records, references, and fixed points. This calculus is defined for your reference on pages 18 to 24.

8. (20 points) Consider the five following class definitions, in the style of TAPL Chapter 18:

```
R = {};  
O = {m:Unit→Unit, n:Unit→Unit, o:Unit→Unit};
```

```
classA = λr:R. λself: Unit→O. λ_:Unit.  
  {m = λ_:Unit. (self unit).n unit,  
   n = λ_:Unit. (self unit).m unit,  
   o = λ_:Unit. unit };
```

```
classB = λr:R. λself: Unit→O. λ_:Unit.  
  let super = classA r self unit in  
  {m = λ_:Unit. unit,  
   n = super.n,  
   o = super.o };
```

```
classC = λr:R. λself: Unit→O. λ_:Unit.  
  let super = classA r self unit in  
  {m = super.m,  
   n = λ_:Unit. super.n unit,  
   o = super.o };
```

```
classD = λr:R. λself: Unit→O. λ_:Unit.  
  let super = classA r self unit in  
  {m = super.m,  
   n = λ_:Unit. super.o unit,  
   o = super.o };
```

```
classE = λr:R. λself: Unit→O. λ_:Unit.  
  let super = classA r self unit in  
  {m = (self unit).m,  
   n = super.n,  
   o = super.o };
```

Each of these classes produces objects with the same type, `O`, which offers three methods. Their state representations are all the trivial record type, `R`. The methods `m`, `n`, and `o` all take `Unit` arguments and return `Unit` results—that is, the only interesting thing about them is whether they diverge or converge. The class `classA` is the “root class” from which the other four are derived.

*continued...*

14 Given these definitions, does evaluation of the following ~~expressions converge (yielding unit) or diverge?~~ <sup>WPE id / last name:</sup> For each one, circle the word “converges” or “diverges,” as appropriate.

<code>((fix (classA {})) unit).n unit</code>	converges	diverges
<code>((fix (classA {})) unit).o unit</code>	converges	diverges
<code>((fix (classB {})) unit).n unit</code>	converges	diverges
<code>((fix (classB {})) unit).o unit</code>	converges	diverges
<code>((fix (classC {})) unit).n unit</code>	converges	diverges
<code>((fix (classC {})) unit).o unit</code>	converges	diverges
<code>((fix (classD {})) unit).n unit</code>	converges	diverges
<code>((fix (classD {})) unit).o unit</code>	converges	diverges
<code>((fix (classE {})) unit).n unit</code>	converges	diverges
<code>((fix (classE {})) unit).o unit</code>	converges	diverges

The definition of Featherweight Java is reproduced on pages 25 to 27.

9. (a) (16 points)

Chapter 3 of TAPL introduced an alternative *big-step* style for presenting evaluation relations, in which each rule shows how to evaluate a term of a particular form “all the way” to a final value, rather than just showing how to make it take a single step. For example, in the lambda-calculus with booleans, the usual small step evaluation rules for *if*...

$$\frac{\begin{array}{l} \text{if true then } t_2 \text{ else } t_3 \rightarrow t_2 \\ \text{if false then } t_2 \text{ else } t_3 \rightarrow t_3 \\ t_1 \rightarrow t'_1 \end{array}}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3}$$

correspond to the following big-step rules:

$$\frac{t_1 \Downarrow \text{true} \quad t_2 \Downarrow v_2}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow v_2}$$

$$\frac{t_1 \Downarrow \text{false} \quad t_3 \Downarrow v_3}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow v_3}$$

*continued...*



Write down a big step evaluation relation for Featherweight Java. (You'll need four inference rules.)

*Grading scheme:*

- 4 points per rule.
- For all rules:
  - -2 points for using a value instead of a term on the LHS.
  - -4 points for a loop in the operational semantics.
  - +/- points for being very close or very wrong.
- Individual rules: Cast:
  - -1 for no subtyping constraint.
  - -2 for no class mentioned in the new expression.
  - -1 for evaluating to wrong type.
- Prj.:
  - -2 for no class mentioned in the record evaluation.
  - -1 for no constraint that project field exists
  - -3 for not projecting from the record
- Meth:
  - -2 for no substitution at all
  - -3 for substitution in the conclusion

*continued...*

*Grading scheme:*

- *No deduction for omitting  $\Gamma$*
- *Either  $v$  or *new*  $C(v)$  accepted for the value*
- *-2 for omitting  $t \Downarrow v$*
- *-3 for neglecting the fact that the new type can be a subtype of old type*
- *hand-wavy english, correct but too vague = 1/5*
- *weird stuff that looks a tiny bit right = 1/5*

## For reference: Simply typed lambda calculus with records (and Nat, Bool, and Unit)

### Syntax

$t ::=$

- unit
- $\{l_i = t_i \mid i \in 1..n\}$
- $t.l$
- $x$
- $\lambda x:T. t$
- $t t$
- true
- false
- if  $t$  then  $t$  else  $t$
- 0
- succ  $t$
- pred  $t$
- iszero  $t$

$v ::=$

- unit
- $\{l_i = v_i \mid i \in 1..n\}$
- $\lambda x:T. t$
- true
- false
- $nv$

$T ::=$

- Unit
- $\{l_i : T_i \mid i \in 1..n\}$
- Bool
- Nat
- $T \rightarrow T$

$\Gamma ::=$

- $\emptyset$
- $\Gamma, x:T$

$nv ::=$

- 0
- succ  $nv$

### terms

*constant unit*  
*record*  
*projection*  
*variable*  
*abstraction*  
*application*  
*constant true*  
*constant false*  
*conditional*  
*constant zero*  
*successor*  
*predecessor*  
*zero test*

### values

*constant unit*  
*record value*  
*abstraction value*  
*true value*  
*false value*  
*numeric value*

### types

*unit type*  
*type of records*  
*type of booleans*  
*type of natural numbers*  
*type of functions*

### contexts

*empty context*  
*term variable binding*

### numeric values

*zero value*  
*successor value*

	$t \rightarrow t'$
$\{\lambda_i = v_i^{i \in 1..n}\} . \lambda_j \rightarrow v_j$	(E-PROJRCD)
$\frac{t_1 \rightarrow t'_1}{t_1 . \lambda \rightarrow t'_1 . \lambda}$	(E-PROJ)
$\frac{t_j \rightarrow t'_j}{\{\lambda_i = v_i^{i \in 1..j-1}, \lambda_j = t_j, \lambda_k = t_k^{k \in j+1..n}\} \rightarrow \{\lambda_i = v_i^{i \in 1..j-1}, \lambda_j = t'_j, \lambda_k = t_k^{k \in j+1..n}\}}$	(E-RCD)
$\frac{t_1 \rightarrow t'_1}{t_1 t_2 \rightarrow t'_1 t_2}$	(E-APP1)
$\frac{t_2 \rightarrow t'_2}{v_1 t_2 \rightarrow v_1 t'_2}$	(E-APP2)
$(\lambda x : T_{11} . t_{12}) v_2 \rightarrow [x \mapsto v_2] t_{12}$	(E-APPABS)
$\text{if true then } t_2 \text{ else } t_3 \rightarrow t_2$	(E-IFTRUE)
$\text{if false then } t_2 \text{ else } t_3 \rightarrow t_3$	(E-IFFALSE)
$\frac{t_1 \rightarrow t'_1}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3}$	(E-IF)
$\frac{t_1 \rightarrow t'_1}{\text{succ } t_1 \rightarrow \text{succ } t'_1}$	(E-SUCC)
$\text{pred } 0 \rightarrow 0$	(E-PREDZERO)
$\text{pred } (\text{succ } n v_1) \rightarrow n v_1$	(E-PREDSUCC)
$\frac{t_1 \rightarrow t'_1}{\text{pred } t_1 \rightarrow \text{pred } t'_1}$	(E-PRED)
$\text{iszero } 0 \rightarrow \text{true}$	(E-ISZEROZERO)
$\text{iszero } (\text{succ } n v_1) \rightarrow \text{false}$	(E-ISZEROSUCC)
$\frac{t_1 \rightarrow t'_1}{\text{iszero } t_1 \rightarrow \text{iszero } t'_1}$	(E-ISZERO)

	$\Gamma \vdash t : T$
$\Gamma \vdash \text{unit} : \text{Unit}$	(T-UNIT)
$\frac{\text{for each } i \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash \{\lambda_i = t_i^{i \in 1..n}\} : \{\lambda_i : T_i^{i \in 1..n}\}}$	(T-RCD)
$\frac{\Gamma \vdash t_1 : \{\lambda_i : T_i^{i \in 1..n}\}}{\Gamma \vdash t_1 . \lambda_j : T_j}$	(T-PROJ)
$\Gamma \vdash \text{true} : \text{Bool}$	(T-TRUE)
$\Gamma \vdash \text{false} : \text{Bool}$	(T-FALSE)
$\frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$	(T-IF)

$\Gamma \vdash 0 : \text{Nat}$	(T-ZERO)
$\frac{\Gamma \vdash t_1 : \text{Nat}}{\Gamma \vdash \text{succ } t_1 : \text{Nat}}$	(T-SUCC)
$\frac{\Gamma \vdash t_1 : \text{Nat}}{\Gamma \vdash \text{pred } t_1 : \text{Nat}}$	(T-PRED)
$\frac{\Gamma \vdash t_1 : \text{Nat}}{\Gamma \vdash \text{iszero } t_1 : \text{Bool}}$	(T-ISZERO)
$\frac{x:T \in \Gamma}{\Gamma \vdash x : T}$	(T-VAR)
$\frac{\Gamma, x:T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x:T_1. t_2 : T_1 \rightarrow T_2}$	(T-ABS)
$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}}$	(T-APP)

## For reference: Subtyping

*New syntactic forms*

$T ::= \dots$   
 $\text{Top}$

*New subtyping rules*

$$S <: S$$

$S <: T$

(S-REFL)

$$\frac{S <: U \quad U <: T}{S <: T}$$

(S-TRANS)

$$S <: \text{Top}$$

(S-TOP)

$$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$$

(S-ARROW)

$$\{l_i : T_i^{i \in 1..n+k}\} <: \{l_i : T_i^{i \in 1..n}\}$$

(S-RCDWIDTH)

$$\frac{\text{for each } i \quad S_i <: T_i}{\{l_i : S_i^{i \in 1..n}\} <: \{l_i : T_i^{i \in 1..n}\}}$$

(S-RCDDEPTH)

$$\frac{\{k_j : S_j^{j \in 1..n}\} \text{ is a permutation of } \{l_i : T_i^{i \in 1..n}\}}{\{k_j : S_j^{j \in 1..n}\} <: \{l_i : T_i^{i \in 1..n}\}}$$

(S-RCDPERM)

*New typing rules*

$$\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T}$$

$\Gamma \vdash t : T$

(T-SUB)

## For reference: References, Fix, and let

### New syntactic forms

$t ::= \dots$   
 $\text{ref } t$   
 $!t$   
 $t := t$   
 $l$   
 $\text{let } x=t \text{ in } t$   
 $\text{fix } t$

$v ::= \dots$   
 $l$

$T ::= \dots$   
 $\text{Ref } T$

$\mu ::= \dots$   
 $\emptyset$   
 $\mu, l = v$

$\Sigma ::= \dots$   
 $\emptyset$   
 $\Sigma, l:T$

### New evaluation rules

$$\boxed{t \mid \mu \rightarrow t' \mid \mu'}$$

$$\frac{t_1 \mid \mu \rightarrow t'_1 \mid \mu'}{t_1 \ t_2 \mid \mu \rightarrow t'_1 \ t_2 \mid \mu'} \quad (\text{E-APP1})$$

$$\frac{t_2 \mid \mu \rightarrow t'_2 \mid \mu'}{v_1 \ t_2 \mid \mu \rightarrow v_1 \ t'_2 \mid \mu'} \quad (\text{E-APP2})$$

$$(\lambda x:T_{11}.t_{12}) \ v_2 \mid \mu \rightarrow [x \mapsto v_2]t_{12} \mid \mu \quad (\text{E-APPABS})$$

$$\{\lambda_i=v_i \ i \in 1..n\}. \lambda_j \mid \mu \rightarrow v_j \mid \mu \quad (\text{E-PROJRCD})$$

$$\frac{t_1 \mid \mu \rightarrow t'_1 \mid \mu'}{t_1. \lambda \mid \mu \rightarrow t'_1. \lambda \mid \mu'} \quad (\text{E-PROJ})$$

$$\frac{t_j \mid \mu \rightarrow t'_j \mid \mu'}{\{\lambda_i=v_i \ i \in 1..j-1, \lambda_j=t_j, \lambda_k=t_k \ k \in j+1..n\} \mid \mu \rightarrow \{\lambda_i=v_i \ i \in 1..j-1, \lambda_j=t'_j, \lambda_k=t_k \ k \in j+1..n\} \mid \mu'} \quad (\text{E-RCD})$$

$$\frac{l \notin \text{dom}(\mu)}{\text{ref } v_1 \mid \mu \rightarrow l \mid (\mu, l \mapsto v_1)} \quad (\text{E-REFV})$$

$$\frac{t_1 \mid \mu \rightarrow t'_1 \mid \mu'}{\text{ref } t_1 \mid \mu \rightarrow \text{ref } t'_1 \mid \mu'} \quad (\text{E-REF})$$

$$\frac{\mu(l) = v}{!l \mid \mu \rightarrow v \mid \mu} \quad (\text{E-DEREFLOC})$$

### terms

*reference creation*  
*dereference*  
*assignment*  
*store location*  
*let binding*  
*fixed point of  $t$*

### values

*store location*

### types

*type of reference cells*

### stores

*empty store*  
*location binding*

### store typings

*empty store typing*  
*location typing*

$\frac{t_1 \mid \mu \rightarrow t'_1 \mid \mu'}{!t_1 \mid \mu \rightarrow !t'_1 \mid \mu'}$	(E-DEREF)
$l := v_2 \mid \mu \rightarrow \text{unit} \mid [l \mapsto v_2] \mu$	(E-ASSIGN)
$\frac{t_1 \mid \mu \rightarrow t'_1 \mid \mu'}{t_1 := t_2 \mid \mu \rightarrow t'_1 := t_2 \mid \mu'}$	(E-ASSIGN1)
$\frac{t_2 \mid \mu \rightarrow t'_2 \mid \mu'}{v_1 := t_2 \mid \mu \rightarrow v_1 := t'_2 \mid \mu'}$	(E-ASSIGN2)
$\text{if true then } t_2 \text{ else } t_3 \mid \mu \rightarrow t_2 \mid \mu$	(E-IFTRUE)
$\text{if false then } t_2 \text{ else } t_3 \mid \mu \rightarrow t_3 \mid \mu$	(E-IFFALSE)
$\frac{t_1 \mid \mu \rightarrow t'_1 \mid \mu'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \mid \mu \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3 \mid \mu'}$	(E-IF)
$\frac{t_1 \mid \mu \rightarrow t'_1 \mid \mu'}{\text{succ } t_1 \mid \mu \rightarrow \text{succ } t'_1 \mid \mu'}$	(E-SUCC)
$\text{pred } 0 \mid \mu \rightarrow 0 \mid \mu$	(E-PREDZERO)
$\text{pred (succ } nv_1) \mid \mu \rightarrow nv_1 \mid \mu$	(E-PREDSUCC)
$\frac{t_1 \mid \mu \rightarrow t'_1 \mid \mu'}{\text{pred } t_1 \mid \mu \rightarrow \text{pred } t'_1 \mid \mu'}$	(E-PRED)
$\text{iszero } 0 \mid \mu \rightarrow \text{true} \mid \mu$	(E-ISZEROZERO)
$\text{iszero (succ } nv_1) \mid \mu \rightarrow \text{false} \mid \mu$	(E-ISZEROSUCC)
$\frac{t_1 \mid \mu \rightarrow t'_1 \mid \mu'}{\text{iszero } t_1 \mid \mu \rightarrow \text{iszero } t'_1 \mid \mu'}$	(E-ISZERO)
$\text{let } x = v_1 \text{ in } t_2 \mid \mu \rightarrow [x \mapsto v_1] t_2 \mid \mu$	(E-LETV)
$\frac{t_1 \mid \mu \rightarrow t'_1 \mid \mu'}{\text{let } x = t_1 \text{ in } t_2 \mid \mu \rightarrow \text{let } x = t'_1 \text{ in } t_2 \mid \mu'}$	(E-LET)
$\begin{aligned} & \text{fix } (\lambda x : T_1. t_2) \mid \mu \\ & \rightarrow [x \mapsto (\text{fix } (\lambda x : T_1. t_2))] t_2 \mid \mu \end{aligned}$	(E-FIXBETA)
$\frac{t_1 \mid \mu \rightarrow t'_1 \mid \mu'}{\text{fix } t_1 \mid \mu \rightarrow \text{fix } t'_1 \mid \mu'}$	(E-FIX)

*New subtyping rules*

$\frac{S_1 <: T_1 \quad T_1 <: S_1}{\text{Ref } S_1 <: \text{Ref } T_1}$	$\boxed{S <: T}$ (S-REF)
--	-----------------------------

*New typing rules*

$\frac{x : T \in \Gamma}{\Gamma \mid \Sigma \vdash x : T}$	$\boxed{\Gamma \mid \Sigma \vdash t : T}$ (T-VAR)
$\frac{\Gamma, x : T_1 \mid \Sigma \vdash t_2 : T_2}{\Gamma \mid \Sigma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2}$	(T-ABS)



$$\frac{\Gamma \mid \Sigma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \mid \Sigma \vdash t_2 : T_{11}}{\Gamma \mid \Sigma \vdash t_1 t_2 : T_{12}} \quad (\text{T-APP})$$

$$\Gamma \mid \Sigma \vdash \text{unit} : \text{Unit} \quad (\text{T-UNIT})$$

$$\frac{\Sigma(l) = T_1}{\Gamma \mid \Sigma \vdash l : \text{Ref } T_1} \quad (\text{T-LOC})$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : T_1}{\Gamma \mid \Sigma \vdash \text{ref } t_1 : \text{Ref } T_1} \quad (\text{T-REF})$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \text{Ref } T_{11}}{\Gamma \mid \Sigma \vdash !t_1 : T_{11}} \quad (\text{T-DEREF})$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \text{Ref } T_{11} \quad \Gamma \mid \Sigma \vdash t_2 : T_{11}}{\Gamma \mid \Sigma \vdash t_1 := t_2 : \text{Unit}} \quad (\text{T-ASSIGN})$$

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x:T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x=t_1 \text{ in } t_2 : T_2} \quad (\text{T-LET})$$

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_1}{\Gamma \vdash \text{fix } t_1 : T_1} \quad (\text{T-FIX})$$

## For reference: FJ

### Syntax

$CL ::=$   
 $\text{class } C \text{ extends } C \{ \bar{C} \bar{F}; K \bar{M} \}$

$K ::=$   
 $C(\bar{C} \bar{F}) \{ \text{super}(\bar{F}); \text{this}.\bar{F}=\bar{F}; \}$

$M ::=$   
 $C m(\bar{C} \bar{x}) \{ \text{return } t; \}$

$t ::=$   
 $x$   
 $t.f$   
 $t.m(\bar{F})$   
 $\text{new } C(\bar{F})$   
 $(C) t$

$v ::=$   
 $\text{new } C(\bar{v})$

*class declarations*

*constructor declarations*

*method declarations*

*terms*

*variable*

*field access*

*method invocation*

*object creation*

*cast*

*values*

*object creation*

### Subtyping

$C <: D$

$$\begin{array}{c}
 C <: C \\
 \frac{C <: D \quad D <: E}{C <: E} \\
 \frac{CT(C) = \text{class } C \text{ extends } D \{ \dots \}}{C <: D}
 \end{array}$$

### Field lookup

$fields(C) = \bar{C} \bar{F}$

$$\begin{array}{c}
 fields(\text{Object}) = \bullet \\
 CT(C) = \text{class } C \text{ extends } D \{ \bar{C} \bar{F}; K \bar{M} \} \\
 \frac{fields(D) = \bar{D} \bar{g}}{fields(C) = \bar{D} \bar{g}, \bar{C} \bar{F}}
 \end{array}$$

### Method type lookup

$mtype(m, C) = \bar{C} \rightarrow C$

$$\begin{array}{c}
 CT(C) = \text{class } C \text{ extends } D \{ \bar{C} \bar{F}; K \bar{M} \} \\
 B m(\bar{B} \bar{x}) \{ \text{return } t; \} \in \bar{M} \\
 \frac{}{mtype(m, C) = \bar{B} \rightarrow B} \\
 CT(C) = \text{class } C \text{ extends } D \{ \bar{C} \bar{F}; K \bar{M} \} \\
 m \text{ is not defined in } \bar{M} \\
 \frac{}{mtype(m, C) = mtype(m, D)}
 \end{array}$$

### Method body lookup

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

$$\frac{CT(C) = \text{class } C \text{ extends } D \{ \bar{C} \bar{f}; \kappa \bar{M} \} \quad B m (\bar{B} \bar{x}) \{ \text{return } t; \} \in \bar{M}}{mbody(m, C) = (\bar{x}, t)}$$

$$\frac{CT(C) = \text{class } C \text{ extends } D \{ \bar{C} \bar{f}; \kappa \bar{M} \} \quad m \text{ is not defined in } \bar{M}}{mbody(m, C) = mbody(m, D)}$$

Valid method overriding

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

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

Evaluation

$$\boxed{t \rightarrow t'}$$

$$\frac{fields(C) = \bar{C} \bar{f}}{(\text{new } C(\bar{v})) . f_i \rightarrow v_i}$$

(E-PROJNEW)

$$\frac{mbody(m, C) = (\bar{x}, t_0)}{(\text{new } C(\bar{v})) . m(\bar{u}) \rightarrow [\bar{x} \rightarrow \bar{u}, \text{this} \mapsto \text{new } C(\bar{v})] t_0}$$

(E-INVKNW)

$$\frac{C <: D}{(D) (\text{new } C(\bar{v})) \rightarrow \text{new } C(\bar{v})}$$

(E-CASTNEW)

$$\frac{t_0 \rightarrow t'_0}{t_0 . f \rightarrow t'_0 . f}$$

(E-FIELD)

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

(E-INVK-RECV)

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

(E-INVK-ARG)

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

(E-NEW-ARG)

$$\frac{t_0 \rightarrow t'_0}{(C) t_0 \rightarrow (C) t'_0}$$

(E-CAST)

Term typing

$$\boxed{\Gamma \vdash t : C}$$

$$\frac{x : C \in \Gamma}{\Gamma \vdash x : C}$$

(T-VAR)

$$\frac{\Gamma \vdash t_0 : C_0 \quad fields(C_0) = \bar{C} \bar{f}}{\Gamma \vdash t_0 . f_i : C_i}$$

(T-FIELD)

$$\frac{\Gamma \vdash t_0 : C_0 \quad mtype(m, C_0) = \bar{D} \rightarrow C \quad \Gamma \vdash \bar{t} : \bar{C} \quad \bar{C} <: \bar{D}}{\Gamma \vdash t_0 . m(\bar{t}) : C}$$

(T-INVK)

$$\frac{\text{fields}(C) = \bar{D} \bar{f} \quad \Gamma \vdash \bar{c} : \bar{C} \quad \bar{C} <: \bar{D}}{\Gamma \vdash \text{new } C(\bar{c}) : C} \quad (\text{T-NEW})$$

$$\frac{\Gamma \vdash t_0 : D \quad D <: C}{\Gamma \vdash (C)t_0 : C} \quad (\text{T-UCAST})$$

$$\frac{\Gamma \vdash t_0 : D \quad C <: D \quad C \neq D}{\Gamma \vdash (C)t_0 : C} \quad (\text{T-DCAST})$$

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

Method typing

M OK in C

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

Class typing

C OK

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