CIS 500 — Software Foundations

Final

(Standard version)

April 30, 2013
Answer key

1. (12 points) Recall the definition of the `fold` function on lists (given in the appendix, page 1). Use it to define a function `sum_list_list` that, given a `list (list nat)`, returns the sum of all the `nat`s in it. E.g.

```
sum_list_list [[1,3],[5,6]] = 15
```

(Note that the first line we've given you starts with `Definition`, not `Fixpoint`: your solution should be non-recursive.)

```
Definition sum_list_list  (l : list (list nat)) : nat :=
```

Answer:

```
  fold (fun z => fun h => fold plus h z) 0 l.
```

Grading scheme: 6 points off if not recursive or missing fold, 3 points off if not accumulating, 1,2 or 3 points off for other errors or redundancies.

2. (12 points) Complete the definition at the bottom of the page of an inductive proposition `count` that counts the number of elements of a list satisfying some predicate `P`. For example, if we define

```
Definition iszero (n : nat) : Prop :=
  n = 0.
```

then the propositions

```
count iszero [] 0
count iszero [1,2,3] 0
count iszero [0,1,2,3] 1
count iszero [1,0,0,2,3,0] 3
```

should all be provable, whereas the propositions

```
count iszero [1,2,3] 3
count iszero [0,0] 4
count iszero [] 1
```

should not be provable.

```
Inductive count {X : Type} (P : X -> Prop) : list X -> nat -> Prop :=
```

Answer:

```
      R0 : count P nil 0
    | Rno : forall l n x, count P l n -> ~(P x) -> count P (x::l) n
    | Ryes : forall l n x, count P l n -> P x -> count P (x::l) (S n).
```

Grading scheme: Roughly 4 points for each correct constructor. 1 or 2 points taken off on each one for minor mistakes.

3. (10 points) For each of the types below, write a Coq expression that has that type. (For example, if the type given were `nat->nat`, you might write `fun x:nat, x+5`.) The definitions of `/\` and `\/` are given in the appendix (page 2) for reference.

(a) `forall X Y, (X -> Y) -> X -> list Y`

Possible answers:

```
fun X (f: X -> Y) (x : X) => [f x]
fun X (f: X -> Y) (x : X) => nil
```

...

(b) `forall (X : Type), (X -> Prop) -> Prop`

Possible answers:

```
fun (X : Type) => fun (P : X -> Prop) => True
fun (X : Type) => fun (P : X -> Prop) => forall x : X, P x
```

...

(c) `forall (X Y : Prop), X -> Y -> (X \/ Y) /\ Y`

Possible answers:

```
fun X Y (HX : X) (HY : Y) =>
  conj (X \/ Y) Y (or_introl X Y HX) HY.
```

...

(d) `forall (X Y W Z : Prop), (X /\ Y) \/ (W /\ Z) -> (X \/ Z)`

Possible answers:

```
fun X Y (H : (X /\ Y) \/ (W /\ Z)) =>
  match H with
  | or_introl HXY => ( match HXY with
                         | conj HX HY => or_introl X Z HX
                       end)
  | or_intror HWZ => ( match HWZ with
                         | conj HW HZ => or_intror X Z HZ
                       end)
  end
```

...

Grading scheme: 2 points for each one of (a) and (b), 3 points for the other ones. One point taken off for forgetting to bind the types, additional points taken off for other mistakes.

4. (5 points) The propositions below concern basic properties of the Imp language. For each proposition, indicate whether it is true or false by circling either T or F.

(a) The `cequiv` relation is transitive.

[T] F

(b) A command that doesn't terminate on any input is equivalent to every other program.

T [F]

(c) If `cequiv c1 c2`, then `cequiv (c1;c2) (c2;c1)`.

[T] F

(d) If `cequiv c1 c2`, then `cequiv (c1;c) (c;c2)` for any `c`.

T [F]

(e) If `cequiv c1 c2` and {{ P }} `c1` {{ Q }}, then {{ P }} `c2` {{ Q }}.

[T] F

Grading scheme: One point for each correct item.

5. (12 points) The following Imp program sets `R` to the sum of the initial values of `X` and `Y` *modulo* the initial value of `Z`:

```
R ::= X + Y;
WHILE Z <= R DO
  R ::= R - Z
END
```

Your task is to fill in assertions to make this a well-decorated program (see page 3 of the appendix), relative to an appropriate and post-condition. Please fill in annotations in the spaces provided below.

```
{{ True                                                  }} ->>
{{ (X + Y) mod Z = (X + Y) mod Z                         }}
R ::= X + Y;
{{ R mod Z = (X + Y) mod Z                               }}
WHILE Z <= R DO
  {{ R mod Z = (X + Y) mod Z /\ Z <= R                   }} ->> (1)
  {{ (R - Z) mod Z = (X + Y) mod Z                       }}
  R ::= R - Z
  {{ R mod Z = (X + Y) mod Z                             }}
END
{{ R mod Z = (X + Y) mod Z /\ ~(Z <= R)                  }} ->> (2)
{{ R = (X + Y) mod Z                                     }}
```

Finally, please mark (by circling the associated arrow `->>` and writing either 1 or 2 or both, as appropriate) any places where the following facts about modular arithmetic are needed to validate the correctness of the decorations:

```
(1)      a mod z = (a + z) mod z

(2)      a < z  ->  (a mod z) = a
```

Grading scheme: One point for each pair of consistent annotations, one point for each correctly applied lemma, plus 4 points for finding the correct loop invariant

6. (10 points) A *triangular number* is one that can be expressed as a sum of the form

$$1 + 2 + 3 + ... + n$$

for some n. In Coq notation:

```
Fixpoint tri (n : nat) : nat :=
  match n with
    0 => 0
  | S n' => n + (tri n')
  end.

Definition triangular (t : nat) : Prop :=
  exists n, t = tri n.
```

The following Imp program will terminate only when the initial value of the variable `T` is a triangular number.

```
{{ True }}
X ::= 0;
A ::= 0;
WHILE A <> T DO
  X ::= X + 1;
  A ::= A + X;
  {{ I }}
END
{{ triangular T }}
```

In the space below, write a suitable invariant for this loop — i.e., an assertion `I` such that, if we add `I` as the annotation at the end of the loop as shown, we can fill in annotations on the rest of the program to make it well decorated. (The spaces for these additional annotations are not shown, and you do not need to provide them.)

Answer: `I = A = tri X`
Grading scheme: 2 to 6 points for wrong invariant. 1 or 2 points off for redundant invariants.

7. (10 points) Suppose that we extend the STLC with a new constant `tzap`...

```
Inductive tm : Type :=
  ...
| tzap : tm.
```

... and add a new rule for the evaluation relation.

```
Inductive step : tm -> tm -> Prop :=
    ...
| ST_Zap : forall t, t ==> tzap.
```

Do the properties below hold of this extension? Write "yes" if the property holds, or give a counterexample if it doesn't.

(a) Progress: *Answer:* Holds

(b) Preservation: *Answer:* Doesn't hold. For instance, `ttrue ==> tzap`, and `empty |- ttrue : Bool`, but it is not the case that `empty |- tzap : Bool`.

(c) `step` is deterministic: *Answer:* Doesn't hold. For instance, take `t = tif ttrue ttrue ttrue`. By `ST_IfTrue`, we have that `t ==> ttrue`, but we also have `t ==> tzap`, and `ttrue <> tzap`.

Grading scheme: 3 points off for each wrong answer. 1 or 2 points off for unclear explanations.

8. (15 points) Suppose L is some variant of STLC with the same syntax but in which the typing and single-step reduction rules have been changed in some way (by adding, removing, or changing rules). Consider the following schematic statement:

> "If the (A) theorem (B) for L, then changing L's (C) relation by (D) a rule might cause it to (E)."

If we substitute "preservation" for (A), "holds" for (B), "typing" for (C), "adding" for (D), and "fail" for (E), we obtain the statement "If the *preservation* theorem *holds* for L, then changing L's *typing* relation by *adding* a rule might cause it to *fail*," which happens to be true: adding a rule to the typing relation of a language like the STLC can sometimes destroy the preservation property.

Fill in the last column of the following table to indicate whether the statement obtained by similarly substituting the values in columns (A) to (E) is true or false. (We've done the first one for you.)

(A)	(B)	(C)	(D)	(E)	T/F
preservation	holds	typing	adding	fail	T
preservation	holds	single-step reduction	removing	fail	F
preservation	holds	typing	removing	fail	T
preservation	holds	single-step reduction	adding	fail	T
progress	holds	typing	removing	fail	F
progress	holds	single-step reduction	removing	fail	T
progress	holds	typing	adding	fail	T
progress	holds	single-step reduction	adding	fail	F
preservation	fails	typing	removing	hold	T
preservation	fails	single-step reduction	removing	hold	T
preservation	fails	typing	adding	hold	T
preservation	fails	single-step reduction	adding	hold	F
progress	fails	typing	removing	hold	T
progress	fails	single-step reduction	removing	hold	F
progress	fails	typing	adding	hold	F
progress	fails	single-step reduction	adding	hold	T

9. (9 points) Name and briefly discuss two advantages of the small-step style over the big-step style of operational semantics.

Answer:

- *The small-step presentation is lower-level and closer to an actual implementation on a concrete machine.*
- *The small-step presentation distinguishes divergence from getting stuck.*
- *The small-step presentation extends naturally to concurrent languages.*
- *A small-step presentation will often be easier to equip with a sensible cost model.*

Grading scheme: The answer was to be given using English, so the grading had to be necessarily a little bit subjective on the our valuation of the student's understanding (when not a perfect answer). In particular, we gave full grade to short but to the point answers, like: "XXX is an advantage because ...." We took as a base the four sample answers above, plus the start of the SmallStep chapter, and in particular we gave full credit for citing the usual theorems associated to small-step as an advantage. We gave two points for "small step style captures the evaluation strategy". When the two advantages given were correct, but we felt that they were so related as to really count as one we graded 5-7. We took away 1 point for slightly misleading statements. We took away 1-3.5 points for more serious errors.

10. (10 points) Consider the following propositions about typing in STLC. For each one, either give values for the existentially quantified variables to make the term typeable, or write "not typeable" and briefly explain why not.

(a) There exist types `T` and `T1` such that

```
empty |- (λx:T. if x then true else x) ∈ T1
```

Answers:
True. E.g., we have `T = bool, T1 = bool -> bool`

(b) For some types `T2` and `T1`,

```
y:T2 |- (λx:T1. x (y (y x) y)) ∈ Bool
```

Answers:
Not typeable.

(c) For all types `T`, there exist types `T1` and `T2` such that

```
y:T2, x:T1 |- (y (x true) x) ∈ T
```

Answers:
`T1 = bool -> bool, T2 : bool -> (bool -> bool) -> T`

(d) For all types `T2` and `T1`, there exists a type `T` such that

```
y:T2 |- (λx:T1. if y x then true else x) ∈ T
```

Possible answers: Not typeable, `T2 = bool, T1 = bool`.

(e) There exist types `T1` and `T2` such that

```
empty |- (λy:T2. λx:T1. if x then y (y (y x)) else x) ∈ Bool
```

Possible answers: Not typeable.

Grading scheme: 2 points for each correct answer. No points for incorrect answer. One point for incomplete answer (Not typable, but not stating the reason.) One point off for other mistakes.

11. (10 points) The Preservation theorem is sometimes called "subject reduction" because, if we think of the term `t` as the subject of the sentence "`t` has type `T` in context Γ," then Preservation amounts to saying that the truth of this sentence is preserved when we reduce the subject.

Conversely, we might wonder about the following property:

```
Theorem subject_expansion : forall (t t' : tm) (T : ty),
    empty |- t' ∈ T ->
    t ==> t' ->
    empty |- t ∈ T
```

Does this property hold in STLC? If so, briefly explain why; otherwise, provide a counterexample. For reference, the original definition of preservation is given in the appendix.
Possible answer: The property does not hold. Consider the following counterexample:

```
t'= true
t = if true then true else \x.x
T = bool
```

Grading scheme: 3 points for wrong counterexample. From 4 to 8 points for correct counterexample but unclear explanation.

12. (5 points) The STLC with records and subtyping is summarized in the appendix for reference (page 10). Indicate whether each of the following claims is true or false for this language.

(a) If `T <: S`, then `(T -> U) <: (S -> U)`.

T [F]

(b) The subtype relation contains an infinite descending chain — that is, there is an infinite sequence of types $\texttt{T}_1$, $\texttt{T}_2$, $\texttt{T}_3$, ... such that, for each i, we have $\texttt{T}_{i+1}\texttt{<:T}_i$ but not $\texttt{T}_i\texttt{<:T}_{i+1}$.

[T] F

(c) There is a record type `T` that is a supertype of every other record type (that is, `S <: T` for every record type `S`).

[T] F

(d) There is a type `T` that is a subtype of every other type (that is, `T <: S` for every type `S`).

T [F]

(e) There is a record type `T` that is a subtype of every other record type (that is, `T <: S` for every record type `S`).

T [F]

Grading scheme: One point for each correct item.

For Reference...

Some functions on lists

```
Fixpoint map {X Y:Type} (f:X->Y) (l:list X) : list Y :=
  match l with
  | []     => []
  | h :: t => (f h) :: (map f t)
  end.

Fixpoint fold {X Y:Type} (f:X->Y->Y) (y:Y) (l:list X) : Y :=
  match l with
  | []     => y
  | h :: t => f h (fold f y t)
  end.
```

Definitions of logical connectives in Coq

```
Inductive and (P Q : Prop) : Prop :=
  conj : P -> Q -> (and P Q).

Inductive or (P Q : Prop) : Prop :=
  | or_introl : P -> or P Q
  | or_intror : Q -> or P Q.

Notation "P /\ Q" := (and P Q) : type_scope.
Notation "P \/ Q" := (or P Q) : type_scope.
```

Formal definitions for Imp

Syntax

```
Inductive aexp : Type :=
  | ANum : nat -> aexp
  | AId : id -> aexp
  | APlus : aexp -> aexp -> aexp
  | AMinus : aexp -> aexp -> aexp
  | AMult : aexp -> aexp -> aexp.

Inductive bexp : Type :=
  | BTrue : bexp
  | BFalse : bexp
  | BEq : aexp -> aexp -> bexp
  | BLe : aexp -> aexp -> bexp
  | BNot : bexp -> bexp
  | BAnd : bexp -> bexp -> bexp.

Inductive com : Type :=
  | CSkip : com
  | CAss : id -> aexp -> com
  | CSeq : com -> com -> com
  | CIf : bexp -> com -> com -> com
  | CWhile : bexp -> com -> com.

Notation "'SKIP'" :=
  CSkip.
Notation "X '::=' a" :=
  (CAss X a) (at level 60).
Notation "c1 ; c2" :=
  (CSeq c1 c2) (at level 80, right associativity).
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c) (at level 80, right associativity).
Notation "'IFB' e1 'THEN' e2 'ELSE' e3 'FI'" :=
  (CIf e1 e2 e3) (at level 80, right associativity).
```

Evaluation relation

```
Inductive ceval : com -> state -> state -> Prop :=
  | E_Skip : forall st,
      SKIP / st || st
  | E_Ass  : forall st a1 n X,
      aeval st a1 = n ->
      (X ::= a1) / st || (update st X n)
  | E_Seq : forall c1 c2 st st' st'',
      c1 / st  || st' ->
      c2 / st' || st'' ->
      (c1 ; c2) / st || st''
  | E_IfTrue : forall st st' b1 c1 c2,
      beval st b1 = true ->
      c1 / st || st' ->
      (IFB b1 THEN c1 ELSE c2 FI) / st || st'
  | E_IfFalse : forall st st' b1 c1 c2,
      beval st b1 = false ->
      c2 / st || st' ->
      (IFB b1 THEN c1 ELSE c2 FI) / st || st'
  | E_WhileEnd : forall b1 st c1,
      beval st b1 = false ->
      (WHILE b1 DO c1 END) / st || st
  | E_WhileLoop : forall st st' st'' b1 c1,
      beval st b1 = true ->
      c1 / st || st' ->
      (WHILE b1 DO c1 END) / st' || st'' ->
      (WHILE b1 DO c1 END) / st || st''

  where "c1 '/' st '||' st'" := (ceval c1 st st').
```

Program equivalence

```
Definition bequiv (b1 b2 : bexp) : Prop :=
  forall (st:state), beval st b1 = beval st b2.

Definition cequiv (c1 c2 : com) : Prop :=
  forall (st st' : state),
    (c1 / st || st') <-> (c2 / st || st').
```

Hoare triples

```
Definition hoare_triple (P:Assertion) (c:com) (Q:Assertion) : Prop :=
  forall st st', c / st || st' -> P st  -> Q st'.

Notation "{{ P }}  c  {{ Q }}" := (hoare_triple P c Q)
                                  (at level 90, c at next level)
                                  : hoare_spec_scope.
```

Implication on assertions

```
Definition assert_implies (P Q : Assertion) : Prop :=
  forall st, P st -> Q st.

Notation "P ⇨ Q" := (assert_implies P Q) (at level 80).
```

Hoare logic rules

$$\frac{}{\{\!\{\texttt{assn\_sub X a } Q \}\!\}\ \texttt{X := a } \{\!\{ Q \}\!\}} \quad (\texttt{hoare\_asgn})$$

$$\frac{}{\{\!\{ P \}\!\}\ \texttt{SKIP } \{\!\{ P \}\!\}} \quad (\texttt{hoare\_skip})$$

$$\frac{\{\!\{ P \}\!\}\ \texttt{c1 } \{\!\{ Q \}\!\} \quad \{\!\{ Q \}\!\}\ \texttt{c2 } \{\!\{ R \}\!\}}{\{\!\{ P \}\!\}\ \texttt{c1; c2 } \{\!\{ R \}\!\}} \quad (\texttt{hoare\_seq})$$

$$\frac{\{\!\{ P \wedge b \}\!\}\ \texttt{c1 } \{\!\{ Q \}\!\} \quad \{\!\{ P \wedge \sim b \}\!\}\ \texttt{c2 } \{\!\{ Q \}\!\}}{\{\!\{ P \}\!\}\ \texttt{IFB b THEN c1 ELSE c2 FI } \{\!\{ Q \}\!\}} \quad (\texttt{hoare\_if})$$

$$\frac{\{\!\{ P \wedge b \}\!\}\ \texttt{c } \{\!\{ P \}\!\}}{\{\!\{ P \}\!\}\ \texttt{WHILE b DO c END } \{\!\{ P \wedge \sim b \}\!\}} \quad (\texttt{hoare\_while})$$

$$\frac{\{\!\{ P' \}\!\}\ \texttt{c } \{\!\{ Q' \}\!\} \quad P \Rightarrow P' \quad Q' \Rightarrow Q}{\{\!\{ P \}\!\}\ \texttt{c } \{\!\{ Q \}\!\}} \quad (\texttt{hoare\_consequence})$$

Decorated programs

A decorated program consists of the program text interleaved with assertions. To check that a decorated program represents a valid proof, we check that each individual command is *locally* consistent with its accompanying assertions in the following sense:

- `SKIP` is locally consistent if its precondition and postcondition are the same:

```
{{ P }}
SKIP
{{ P }}
```

- The sequential composition of commands `c1` and `c2` is locally consistent (with respect to assertions P and R) if `c1` is locally consistent (with respect to P and Q) and `c2` is locally consistent (with respect to Q and R):

```
{{ P }}
c1;
{{ Q }}
c2
{{ R }}
```

- An assignment is locally consistent if its precondition is the appropriate substitution of its postcondition:

```
{{ P where a is substituted for X }}
X ::= a
{{ P }}
```

- A conditional is locally consistent (with respect to assertions P and Q) if the assertions at the top of its "then" and "else" branches are exactly $P \wedge$ `b` and $P \wedge \sim$`b` and if its "then" branch is locally consistent (with respect to $P \wedge$ `b` and Q) and its "else" branch is locally consistent (with respect to $P \wedge \sim$`b` and Q):

```
{{ P }}
IFB b THEN
  {{ P ∧ b }}
  c1
  {{ Q }}
ELSE
  {{ P ∧ ~b }}
  c2
  {{ Q }}
FI
{{ Q }}
```

- A while loop is locally consistent if its postcondition is $P \wedge \sim$`b` (where P is its precondition) and if the pre- and postconditions of its body are exactly $P \wedge$ `b` and P:

```
{{ P }}
WHILE b DO
  {{ P ∧ b }}
  c1
  {{ P }}
END
{{ P ∧ ~b }}
```

- A pair of assertions separated by `=>` is locally consistent if the first implies the second (in all states):

  `{{` P `}} ->>`
  `{{` Q `}}`

STLC with booleans

Syntax

```
T ::= Bool                t  ::= x                        v ::= true
    | T -> T                   | t t                          | false
                               | \x:T. t                      | \x:T. t
                               | true
                               | false
                               | if t then t else t
```

Small-step operational semantics

$$\frac{\text{value } v2}{(\lambda x{:}T.t12)\ v2 \Longrightarrow [x:=v2]t12}\ (\text{ST\_AppAbs})$$

$$\frac{t1 \Longrightarrow t1'}{t1\ t2 \Longrightarrow t1'\ t2}\ (\text{ST\_App1})$$

$$\frac{\text{value } v1 \quad t2 \Longrightarrow t2'}{v1\ t2 \Longrightarrow v1\ t2'}\ (\text{ST\_App2})$$

$$\frac{}{(\text{if true then } t1 \text{ else } t2) \Longrightarrow t1}\ (\text{ST\_IfTrue})$$

$$\frac{}{(\text{if false then } t1 \text{ else } t2) \Longrightarrow t2}\ (\text{ST\_IfFalse})$$

$$\frac{t1 \Longrightarrow t1'}{(\text{if } t1 \text{ then } t2 \text{ else } t3) \Longrightarrow (\text{if } t1' \text{ then } t2 \text{ else } t3)}\ (\text{ST\_If})$$

Typing

$$\frac{\Gamma\ \texttt{x} = \texttt{T}}{\Gamma \vdash \texttt{x} \in \texttt{T}} \quad \text{(T\_Var)}$$

$$\frac{\Gamma,\ \texttt{x:T11} \vdash \texttt{t12} \in \texttt{T12}}{\Gamma \vdash \texttt{\textbackslash x:T11.t12} \in \texttt{T11->T12}} \quad \text{(T\_Abs)}$$

$$\frac{\Gamma \vdash \texttt{t1} \in \texttt{T11->T12} \qquad \Gamma \vdash \texttt{t2} \in \texttt{T11}}{\Gamma \vdash \texttt{t1 t2} \in \texttt{T12}} \quad \text{(T\_App)}$$

$$\frac{}{\Gamma \vdash \texttt{true} \in \texttt{Bool}} \quad \text{(T\_True)}$$

$$\frac{}{\Gamma \vdash \texttt{false} \in \texttt{Bool}} \quad \text{(T\_False)}$$

$$\frac{\Gamma \vdash \texttt{t1} \in \texttt{Bool} \qquad \Gamma \vdash \texttt{t2} \in \texttt{T} \qquad \Gamma \vdash \texttt{t3} \in \texttt{T}}{\Gamma \vdash \texttt{if t1 then t2 else t3} \in \texttt{T}} \quad \text{(T\_If)}$$

Properties of STLC

```
Theorem preservation :
  forall t t' T,
    empty |- t ∈ T ->
    t ==> t' ->
    empty |- t' ∈ T.

Theorem progress :
  forall t T,
    empty |- t ∈ T ->
    value t \/ exists t', t ==> t'.
```

STLC with booleans, records and subtyping

Syntax

```
T ::= ...                  t  ::= ...                        v ::= ...
    | {i1:T1, ..., in:Tn}      | {i1=t1, ..., in=tn}             | {i1=v1, ..., in=vn}
    | Top                      | t.i
```

Small-step operational semantics

$$\frac{\texttt{ti ==> ti'}}{\texttt{\{i1=v1, ..., im=vm, in=ti, ...\}} \Longrightarrow \texttt{\{i1=v1, ..., im=vm, in=ti', ...\}}} \quad \text{(ST\_Rcd)}$$

$$\frac{\texttt{t1 ==> t1'}}{\texttt{t1.i ==> t1'.i}} \quad \text{(ST\_Proj1)}$$

$$\frac{}{\texttt{\{..., i=vi, ...\}.i => vi}} \quad \text{(ST\_ProjRcd)}$$

Typing

$$\frac{\Gamma \vdash \texttt{t1} \in \texttt{T1} \quad \ldots \quad \Gamma \vdash \texttt{tn} \in \texttt{Tn}}{\Gamma \vdash \texttt{\{i1=t1,...,in=tn\}} \in \texttt{\{i1:T1,...,in:Tn\}}} \quad \text{(T\_Rcd)}$$

$$\frac{\Gamma \vdash \texttt{t} \in \texttt{\{...,i:T,...\}}}{\Gamma \vdash \texttt{t.i} \in \texttt{T}} \quad \text{(T\_proj)}$$

$$\frac{\Gamma \vdash \texttt{t} \in \texttt{S} \qquad \texttt{S <: T}}{\Gamma \vdash \texttt{t} \in \texttt{T}} \quad \text{(T\_Sub)}$$

Subtyping

$$\frac{S <: U \quad U <: T}{S <: T} \text{ (S\_Trans)}$$

$$\frac{}{T <: T} \text{ (S\_Refl)}$$

$$\frac{}{S <: \mathrm{Top}} \text{ (S\_Top)}$$

$$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \text{ (S\_Arrow)}$$

$$\frac{n > m}{\{i_1 : T_1 \ldots i_n : T_n\} <: \{i_1 : T_1 \ldots i_m : T_m\}} \text{ (S\_RcdWidth)}$$

$$\frac{S_1 <: T_1 \quad \ldots \quad S_n <: T_n}{\{i_1 : S_1 \ldots i_n : S_n\} <: \{i_1 : T_1 \ldots i_n : T_n\}} \text{ (S\_RcdDepth)}$$

$$\frac{\{i_1 : S_1 \ldots i_n : S_n\} \text{ is a permutation of } \{i_1 : T_1 \ldots i_n : T_n\}}{\{i_1 : S_1 \ldots i_n : S_n\} <: \{i_1 : T_1 \ldots i_n : T_n\}} \text{ (S\_RcdPerm)}$$