Before beginning the exam, please write your WPE-I id or PennKey (login ID) at the top of each even-numbered page (so that we can find things if a staple fails!).

Solutions

[Standard Track Only] Types of Coq terms (10 points)

For each Coq term below, give its type or write “ill-typed” if it has no type.

1.1 \( \text{fun} \ (X : \text{Type}) \Rightarrow \text{fun} \ (f : X \rightarrow X) \Rightarrow \text{fun} \ (a : X) \Rightarrow f \ (f \ (f \ a)) \)
   \(\text{Answer:} \ \forall X, X \rightarrow X\)

1.2 \( \text{if true then 5 else false} \)
   \(\text{Answer:} \ \text{ill-typed}\)

1.3 \( \text{fun} \ x \Rightarrow x \ + \ x = 1 \)
   \(\text{Answer:} \ \text{nat} \rightarrow \text{Prop}\)

1.4 \( \forall \text{st}, \text{st} =\{X := Y\} \Rightarrow \text{st} \)
   \(\text{Answer:} \ \text{Prop}\)

1.5 \( \text{(fun} \ P \Rightarrow P \lor \neg P) \text{ False} \)
   \(\text{Answer:} \ \text{Prop}\)
[Standard Track Only] Coq programming (12 points)

For each type below, either give a term that has that type or write “uninhabited.”

2.1 \(\forall (X : \text{Type}), X \to (X \times \text{list} \ X)\)
Answer: \(\text{fun } X \Rightarrow \text{fun } x : X \Rightarrow (x, [])\)

2.2 \(\text{bool} \to \text{Prop}\)
Answer: \(\text{fun } b \Rightarrow b = \text{true}\)

2.3 \(\forall (x : \text{bool}), \text{Prop}\)
Answer: \(\text{fun } b \Rightarrow b = \text{true}\)

2.4 \(\text{False} \to \text{bool}\)
Answer: \(\text{fun } \_ \Rightarrow \text{true}\)

2.5 \(2 + 2 = 5\)
Answer: uninhabited
### Loop invariants (9 points)

For each pair of Hoare triple and proposed loop invariant $\text{Inv}$, your job is to decide whether $\text{Inv}$ can be used to prove a Hoare triple of this form:

$$\{\{ P \} \} \text{WHILE} \; b \; \text{DO} \; c \; \text{END} \; \{\{ Q \} \}$$

Specifically, you should decide whether $\text{Inv}$ satisfies each of the three specific constraints from the Hoare rule for WHILE:

1. **Implied by precondition:** $P \rightarrow \text{Inv}$
2. **Preserved by loop body (when loop guard true):** $\{\{ \text{Inv} \cup b \} \} \; c \; \{\{ \text{Inv} \} \}$
3. **Implies postcondition (when loop guard false):** $(\text{Inv} \cup \neg b) \rightarrow Q$

We call them “Implied by Pre,” “Preserved,” and “Implies Post” below, for brevity.

#### 3.1

$$\{\{ X = 10 \} \}$$

WHILE $Y < 3$ DO

\[
\begin{align*}
X &:= X \cdot 10; \\
Y &:= Y + 1
\end{align*}
\]
END

$$\{\{ X = 1000 \} \}$$

<table>
<thead>
<tr>
<th>Proposed Inv</th>
<th>Implied by Pre</th>
<th>Preserved</th>
<th>Implies Post</th>
</tr>
</thead>
<tbody>
<tr>
<td>exists ($x : \text{nat}$), $X = 10 \cdot x$</td>
<td>☒</td>
<td>☒</td>
<td>☐</td>
</tr>
<tr>
<td>$X \leq 1000$</td>
<td>☒</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>$X = 10 \cdot Y$</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>

#### 3.2

$$\{\{ X = 10 \cup Y = 2 \cup Z = 0 \} \}$$

WHILE $!(X = 0)$ DO

\[
\begin{align*}
X &:= X - Y; \\
Z &:= Z + 1
\end{align*}
\]
END

$$\{\{ Z = 5 \} \}$$

<table>
<thead>
<tr>
<th>Proposed Inv</th>
<th>Implied by Pre</th>
<th>Preserved</th>
<th>Implies Post</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10 = X + Y \cdot Z \cup Y = 2$</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>$X = Z \cdot 2$</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>$X = 10 + Y \cdot Z \cup Y = 2$</td>
<td>☒</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>
Recall the language $HImp$ from the $Equiv$ chapter of $Programming\ Language\ Foundations$. Beginning from the standard Imp language, we add one new form of command
\[
c ::= \ldots \\
HAVOC\ X
\]
and one new clause to the $ceval$ relation:
\[
|\ E_{Havoc} : \forall (st : \text{state}) (X : \text{string}) (n : \text{nat}), \\
\quad st =\left[\ HAVOC\ X \ \Rightarrow\ (X \rightarrow\ n ;\ st)\right]
\]
That is, $HAVOC\ X$ nondeterministically sets $X$ to any value. For example, suppose $c$ is this program:
\[
\begin{align*}
\text{WHILE} & \neg(X .\leq Y) \ \text{DO} \\
& \quad HAVOC\ X;; \\
& \quad Y ::= Y .+ 1 \\
\text{END}
\end{align*}
\]
If we start $c$ in a state $st$ where $X$ is already less than or equal to $Y$, then it will (deterministically) terminate in the same state. On the other hand, if we start $c$ in a state where $X$ is greater than $Y$ then it is possible for $c$ to terminate in any state $st'$ where (1) the final value of $X$ in $st'$ is less than or equal to the final value of $Y$, (2) the final value of $Y$ is greater than its starting value in $st$, and (3) the final values of the other variables are the same as their starting values.

In the space below, write an appropriate Hoare Logic rule for reasoning about $HAVOC$ statements. Your rule should have the form
\[
\text{-----------------------} \\
\{\{ P \}\} \ HAVOC\ X \{\{ Q \}\}
\]
for a suitable precondition $P$ and postcondition $Q$.

$Answer:$
\[
\text{-----------------------} \\
\{\{ \forall n, Q (X \rightarrow\ n ;\ st) \}\} \ HAVOC\ X \{\{ Q \}\}
\]
Loop invariants with Havoc (15 points)

Now let’s consider Hoare triples for the HImp language. For each triple below, find an invariant for the WHILE loop that will allow us to prove the triple. As usual, m and n are arbitrary (but fixed) values of type nat.

5.1  
\{\{ X = m /\ Y = n \}\}
WHILE !(X .= Y) DO
  HAVOC Z;;
  IFB Z .<= Y THEN
    X ::= X .+ Z;;
    Y ::= Y .- Z
  ELSE SKIP FI
\{\{ Invariant goes here \}\}
END
\{\{ 2 * X = m + n \}\}

Answer: Invariant = X + Y = m + n

5.2  
\{\{ X <= 1 \}\}
  Y ::= 0;;
WHILE Z .= 0 DO
  X ::= X .+ Y;;
  HAVOC Z
\{\{ Invariant goes here \}\}
END
\{\{ X <= 1 \}\}

Answer: Invariant = X <= 1 /\ Y = 0

5.3  
\{\{ X = m \}\}
  WHILE X .<= m DO
    HAVOC Z;;
    IFB Z .= 0 THEN
      X ::= X .+ X
    ELSE SKIP FI
\{\{ Invariant goes here \}\}
END
\{\{ X = 2 * m \}\}

Answer: Invariant = (Z <> 0 /\ X = m) /\ (Z = 0 /\ X = 2 * m)
Suppose we define a refinement relation between HImp programs as follows:

Program $c$ refines program $d$ if $st = \{ c \} \Rightarrow st'$ implies $st = \{ d \} \Rightarrow st'$ for all $st$ and $st'$.

This is similar to the definition of $cequiv$ that we studied in the Equiv chapter (repeated on page 4 for reference), but an “iff” has been replaced by an implication (i.e., saying $c_1$ refines $c_2$ and $c_2$ refines $c_1$ is the same as saying that $c_1$ and $c_2$ are equivalent).

6.1 If $c_1$ refines $c_2$, does it follow that $c_1$ and $c_2$ terminate on the same set of initial memory states?

☐ Yes ☒ No

6.2 $c_1 = \text{HAVOC } X; \text{HAVOC } Y$
$c_2 = \text{HAVOC } X; Y ::= X$

☐ $c_1$ refines $c_2$ ☒ $c_2$ refines $c_1$ ☐ Both ☐ Neither

6.3 $c_1 = \text{WHILE true DO SKIP END}$
$c_2 = \text{WHILE } X ::= 0 \text{ DO SKIP END}$

☒ $c_1$ refines $c_2$ ☐ $c_2$ refines $c_1$ ☐ Both ☐ Neither

6.4 $c_1 = \text{HAVOC } X$
$c_2 = \text{HAVOC } X;; X ::= X + 1$

☐ $c_1$ refines $c_2$ ☒ $c_2$ refines $c_1$ ☐ Both ☐ Neither

6.5 $c_1 = \text{HAVOC } X;; \text{WHILE } X > 0 \text{ DO } X ::= X - 1 \text{ END}$
$c_2 = X ::= 100;; \text{WHILE } X > 0 \text{ DO HAVOC } X \text{ END}$

☐ $c_1$ refines $c_2$ ☒ $c_2$ refines $c_1$ ☐ Both ☐ Neither

6.6 Let $P$ be the set of HImp programs that mention the variables $X$, $Y$, and $Z$ (or a subset of these, but no others). Does $P$ have a minimal element in the refinement ordering—i.e., is there some $p \in P$ such that $p$ refines $c$ for every $c \in P$?

☒ Yes ☐ No

If so, then what is an example of such a minimal element?

Answer: Any nonterminating program.

6.7 Does the set $P$ have any maximal elements (i.e., any elements that are refined by every element of $P$)?

☒ Yes ☐ No

If so, then give an example.

Answer: HAVOC $X;;$ HAVOC $Y;;$ HAVOC $Z$
Give a detailed informal proof of the following theorem about the refinement relation from the previous problem. Please be sure to state your induction hypothesis explicitly at the beginning of each inductive case.

**Theorem:** If $c$ refines $d$, then $\text{WHILE } b \text{ DO } c \text{ END}$ refines $\text{WHILE } b \text{ DO } d \text{ END}$.

**Proof:** Suppose $st = [\text{WHILE } b \text{ DO } c \text{ END}] \Rightarrow st'$. Proceed by induction on this derivation. (To be completely pedantic, we are proving the proposition for all $b, c$, if $c_0 = \text{WHILE } b \text{ DO } c \text{ END}$, then $st = [\text{WHILE } b \text{ DO } d \text{ END}] \Rightarrow st'$ by induction on a derivation of $st = [c_0] \Rightarrow st'$.)

- **Case $E_{\text{WhileFalse}}$:** Then $st \ b = \text{false}$ and $st' = st$. By $\text{ST}_{\text{WhileFalse}}$, $st = [\text{WHILE } b \text{ DO } d \text{ END}] \Rightarrow st'$ as required.

- **Case $E_{\text{WhileTrue}}$:** Then
  
  (a) $st \ b = \text{true}$
  (b) $st = [c] \Rightarrow st'$
  (c) $st' = [\text{WHILE } b \text{ DO } c \text{ END}] \Rightarrow st''$

  and we have this IH:

  $st' = [\text{WHILE } b \text{ DO } d \text{ END}] \Rightarrow st''$

  But $c$ refines $d$, so by (b), $st = [d] \Rightarrow st'$. Then, by $E_{\text{WhileTrue}}$, we have $st = [\text{WHILE } b \text{ DO } d \text{ END}] \Rightarrow st'$ as required.
The Logic chapter of *Logical Foundations* introduced the axiom of *functional extensionality*.

8.1 State this axiom (in formal Coq notation, if possible, otherwise in words).

Axiom functional_extensionality :
  forall {X Y : Type} {f g : X -> Y},
  (forall (x : X), f x = g x) -> f = g.

8.2 Briefly (1-3 sentences) explain how it is used in LF (e.g., in the treatment of total and partial maps in chapter Maps).

*Answer*: Adopting this axiom allows us to state and use properties of maps in a simple and direct way. For example, with functional extensionality we can write

Theorem t_update_same : forall (A : Type) (m : total_map A) x,
  (x !-> m x ; m) = m.

instead of this:

Theorem t_update_same_bad : forall (A : Type) (m : total_map A) x y,
  (x !-> m x ; m) y = m y.
Recall the familiar Fibonacci function:

```plaintext
fib x = 
  if x = 0 then 0 
  else if x = 1 then 1 
  else fib (x-2) + fib (x-1)
```

We saw in chapter MoreStlc how adding a fix primitive (see page 9 in the handout) to the STLC allows us to write such definitions in a more basic way.

Use fix to finish the following definition (without mentioning the identifier fib) in the STLC with numbers, booleans, and fix.

```plaintext
fib = 
  fix 
    (\f:Nat->Nat.
     (\x:Nat.
      if x=0 then 0 else if x=1 then 1 
      else f (x-2) + f (x-1))
```
Substitution as a relation (12 points)

Recall the definition of the substitution operation in the STLC extended with let (pages 6, 7) (but omitting base types such as booleans for brevity):

\[
\text{Fixpoint subst } (x : \text{string}) \ (s : \text{tm}) \ (t : \text{tm}) : \text{tm} := \\
\text{match } t \text{ with} \\
\quad \text{| var } y \Rightarrow \\
\quad \quad \text{if eqb_string } x \ y \text{ then } s \text{ else } t \\
\quad \text{| abs } y \ T \ t1 \Rightarrow \\
\quad \quad \text{abs } y \ T \ (\text{if eqb_string } x \ y \text{ then } t1 \text{ else } \text{subst } x \ s \ t1) \\
\quad \text{| app } t1 \ t2 \Rightarrow \\
\quad \quad \text{app } (\text{subst } x \ s \ t1) \ (\text{subst } x \ s \ t2) \\
\quad \text{| tlet } y \ t1 \ t2 \Rightarrow \\
\quad \quad \text{tlet } y \ (\text{subst } x \ s \ t1) \ (\text{if eqb_string } x \ y \text{ then } t2 \text{ else } \text{subst } x \ s \ t2) \\
\text{end.}
\]

This definition uses Coq’s Fixpoint facility to define substitution as a function. Suppose, instead, we wanted to define substitution as an inductive relation substi. We’ve begun the definition by providing the Inductive header and one of the constructors; your job is to fill in the rest of the constructors. (Your answer should have the property that subst x s t = t’ <-> substi x s t t’, for all s, x, t, and t’, but you do not need to prove it).

Answer:

\[
\text{Inductive substi } (x : \text{id}) \ (s : \text{tm}) : \text{tm} \rightarrow \text{tm} \rightarrow \text{Prop} := \\
\quad \text{| s_app } : \forall t1 \ t2 \ t1’ \ t2’, \\
\quad \quad \text{substi } x \ s \ t1 \ t1’ \rightarrow \\
\quad \quad \text{substi } x \ s \ t2 \ t2’ \rightarrow \\
\quad \quad \text{substi } x \ s \ (\text{app } t1 \ t2) \ (\text{app } t1’ \ t2’) \\
\quad \text{| s_var1 } : \\
\quad \quad \text{substi } x \ s \ (\text{var } x) \ s \\
\quad \text{| s_var2 } : \forall x’, \\
\quad \quad \text{eqb_string } x \ x’ = \text{false} \rightarrow \\
\quad \quad \text{substi } s \ x \ (\text{var } x’) \ (\text{var } x’) \\
\quad \text{| s_abs1 } : \forall T \ t1, \\
\quad \quad \text{substi } s \ x \ (\text{abs } x \ T \ t1) \ (\text{abs } x \ T \ t1) \\
\quad \text{| s_abs2 } : \forall x’ \ T \ t1 \ t1’, \\
\quad \quad \text{eqb_string } x \ x’ = \text{false} \rightarrow \\
\quad \quad \text{substi } x \ s \ t1 \ t1’ \rightarrow \\
\quad \quad \text{substi } x \ s \ (\text{abs } x’ \ T \ t1) \ (\text{abs } x’ \ T \ t1’) \\
\quad \text{| s_let1 } : \forall t1 \ t2 \ t1’, \\
\quad \quad \text{substi } x \ s \ t1 \ t1’ \rightarrow \\
\quad \quad \text{substi } x \ s \ (\text{tlet } x \ t1’ \ t2) \\
\quad \text{| s_let2 } : \forall x \ y \ t1 \ t2 \ t1’ \ t2’, \\
\quad \quad \text{eqb_string } x \ y = \text{false} \rightarrow \\
\quad \quad \text{substi } x \ s \ t1 \ t1’ \rightarrow \\
\quad \quad \text{substi } x \ s \ t2 \ t2’ \rightarrow \\
\quad \quad \text{substi } x \ s \ (\text{tlet } y \ t1’ \ t2’). \\
\]
Properties of reduction and typing (12 points)

Suppose we are given some new programming language — i.e., someone specifies

- a set of terms \( \text{tm} \),
- a property \( \text{value} : \text{tm} \rightarrow \text{Prop} \) that picks out a subset of terms designated as values,
- a small-step reduction relation \( \text{stepsto} : \text{tm} \rightarrow \text{tm} \rightarrow \text{Prop} \),
- a set of types \( \text{ty} \), and
- a typing relation \( \text{has_type} : \text{tm} \rightarrow \text{ty} \rightarrow \text{Prop} \).

For simplicity, let’s suppose that this language doesn’t have any variable binders, so we don’t need any contexts, and typing is just a two-place relation. Please also assume that the typing relation makes no mention of the step relation, and vice versa.

We use lower-case variables like \( t \) to stand for terms (in \( \text{tm} \)) and upper-case variables like \( T \) to stand for types (in \( \text{ty} \)). We write \( t_1 \rightarrow t_2 \) to mean \( \text{stepsto} \ t_1 \ t_2 \) and \( \vdash t \ \in T \) to mean \( \text{has_type} \ t \ T \).

Further, suppose that we are told the following facts about this language:

- Uniqueness of typing: If \( t \) is a term and \( T_1, T_2 \) are types such that \( \vdash t \ \in T_1 \) and \( \vdash t \ \in T_2 \), then \( T_1 = T_2 \).
- Determinism of reduction: If \( t \rightarrow t_1 \) and \( t \rightarrow t_2 \), then \( t_1 = t_2 \).
- Progress: If \( \vdash t \ \in T \), then either \( t \) is a value or else there is some \( t' \) such that \( t \rightarrow t' \).
- Preservation: If \( \vdash t \ \in T \) and \( t \rightarrow t' \), then \( \vdash t' \ \in T \).
- Values are normal forms: If \( t \) is a value, then there is no \( t' \) such that \( t \rightarrow t' \).

In each of the following parts, we ask you to consider how a proposed change to this language will affect these properties (without knowing anything more about the details of the language). If the proposed change will definitely break the corresponding property, check the box by “fails.” If the proposed change definitely cannot break the property, check the box next to “holds.” If this change might or might not break the property, depending on the details of the original language and/or exactly what is added or removed, choose “depends.”
11.1 If we restrict the typing relation (i.e., we take one or more pairs of a term \( t \) and a type \( T \) with \( |- t \in T \) and remove them from the relation), what happens to these properties?

<table>
<thead>
<tr>
<th>Property</th>
<th>( \nabla ) holds</th>
<th>( \Box ) fails</th>
<th>( ☐ ) depends</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniqueness of typing</td>
<td>( \nabla )</td>
<td>( \Box )</td>
<td>( ☐ )</td>
</tr>
<tr>
<td>Determinism of reduction</td>
<td>( \nabla )</td>
<td>( \Box )</td>
<td>( ☐ )</td>
</tr>
<tr>
<td>Progress</td>
<td>( \nabla )</td>
<td>( \Box )</td>
<td>( ☐ )</td>
</tr>
<tr>
<td>Preservation</td>
<td>( \nabla )</td>
<td>( \Box )</td>
<td>( ☐ )</td>
</tr>
</tbody>
</table>

Typing only gets “more unique.” Reduction is not affected. Progress is easy because it mentions the typing relation only in a hypothesis, so restricting typing obviously makes the relation “more true.” Preservation, though, has typing on both sides of an arrow. So, for example, if we remove the pair \( |- true \in Bool \) from the typing relation, then the term \( if true then true else true \) will still be well typed, but it will reduce to \( true \), which is not. Conversely, if we remove all pairs from the typing relation (leaving it empty), then preservation still holds (trivially).

11.2 If we enlarge the typing relation (i.e., we add one or more new pairs \( |- t \in T \) that were not in the original relation), what happens?

<table>
<thead>
<tr>
<th>Property</th>
<th>( \nabla ) holds</th>
<th>( \Box ) fails</th>
<th>( ☐ ) depends</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniqueness of typing</td>
<td>( \nabla )</td>
<td>( \Box )</td>
<td>( ☐ )</td>
</tr>
<tr>
<td>Determinism of reduction</td>
<td>( \nabla )</td>
<td>( \Box )</td>
<td>( ☐ )</td>
</tr>
<tr>
<td>Progress</td>
<td>( \nabla )</td>
<td>( \Box )</td>
<td>( ☐ )</td>
</tr>
<tr>
<td>Preservation</td>
<td>( \nabla )</td>
<td>( \Box )</td>
<td>( ☐ )</td>
</tr>
</tbody>
</table>

Uniqueness of typing might either fail (if we add a new pair \( |- t \in T \) where \( t \) was already well typed (but with a different type \( T \)) or continue to hold (if we only add pairs \( |- t \in T \) where \( t \) was not typeable before). Determinism of reduction is unaffected by changes to typing. Progress may fail (if a non-value normal form is given a type) or not. Preservation may fail (if we give a type to a non-typeable term that also reduces to a non-typeable term) or not.

11.3 If we restrict the reduction relation for well-typed terms (i.e., we take one or more pairs of terms \( t \) and \( t' \) with \( t \rightarrow t' \) and \( |- t \in T \) for some type \( T \) and remove them from the reduction relation), what happens?

<table>
<thead>
<tr>
<th>Property</th>
<th>( \nabla ) holds</th>
<th>( \Box ) fails</th>
<th>( ☐ ) depends</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniqueness of typing</td>
<td>( \nabla )</td>
<td>( \Box )</td>
<td>( ☐ )</td>
</tr>
<tr>
<td>Determinism of reduction</td>
<td>( \nabla )</td>
<td>( \Box )</td>
<td>( ☐ )</td>
</tr>
<tr>
<td>Progress</td>
<td>( \nabla )</td>
<td>( \Box )</td>
<td>( ☐ )</td>
</tr>
</tbody>
</table>
Preservation: ☒ holds ☐ fails ☐ depends

Changing reduction doesn't affect uniqueness of typing. Reduction will continue to be deterministic if we only restrict it. Progress must fail because we've assumed that values are already normal forms. Preservation continues to hold because this change only makes its premise harder to satisfy.

If we enlarge the reduction relation for well-typed terms (i.e., we add one or more new pairs \( t \rightarrow t' \), where \( |- t \ \text{in} \ T \) is in the original typing relation for some type \( T \)), what happens?

Uniqueness of typing: ☒ holds ☐ fails ☐ depends

Determinism of reduction: ☐ holds ☐ fails ☒ depends

Progress: ☒ holds ☐ fails ☐ depends

Preservation: ☐ holds ☐ fails ☒ depends

Changing reduction doesn't affect uniqueness of typing. Reduction may either continue to be deterministic (if we only add reduction steps for terms that were previously values) or fail. Progress will continue to hold because this change only makes its right-hand side easier to satisfy. Preservation may either continue to hold (if we only add pairs \( t \rightarrow t' \) where \( t \) and \( t' \) both have the same type or are both not typable).
Progress and Preservation for STLC with Subtyping (12 points)

The syntax, operational semantics, and typing rules for the simply-typed lambda calculus with booleans, products, records, and subtyping are given in the Appendix (pages 6, 8, 9, 10).

For each variant below, indicate which of the properties of the original system remain true in the presence of this rule. If a property becomes false, give a counterexample.

12.1 Suppose that we add the following reduction rule:

\[\text{false} \rightarrow \lambda x:\text{Bool}. \ x\]

• Progress ☐ Remains true □ Becomes false

   Explanation: Enlarging the reduction relation can never break progress.

• Preservation □ Remains true ☐ Becomes false

   Explanation: The term \text{false} has type \text{Bool} but steps to \lambda x:\text{Bool}. \ x, which does not have type \text{Bool}.

12.2 Suppose instead that we add the following subtyping rule:

\[\text{(Bool \rightarrow T)} <: T\]

• Progress □ Remains true ☐ Becomes false

   Explanation: \((\lambda x:\text{Bool}. \ (x, x)).\text{fst}\) becomes typeable but it is stuck.

• Preservation □ Remains true ☐ Becomes false

   Explanation: ((\lambda x:\text{Bool}. \ y:{}. \ x) {}) can be given type \text{Bool} but steps to \lambda y:{}. \ {}, which can only have an arrow type (or \text{Top}).

12.3 Suppose, instead, that we consider a variant in which we add a new term \text{wrong} with the following reduction rule (and no typing rules):

\[\text{wrong} \rightarrow \text{wrong wrong}\]

• Progress ☐ Remains true □ Becomes false

   Explanation: We are only interested in progress of well-typed terms, and such terms cannot contain \text{wrong}.

• Preservation ☐ Remains true □ Becomes false

   Explanation: We are only interested in progress of well-typed terms, and such terms cannot contain \text{wrong}.
13. **Subtyping and typechecking** (16 points)

13.1

\[ \text{empty |- } (\forall p: T. \{x=p.x, y=p.y\}) \{x=\text{true}, y=\text{unit}, z=\text{true}\} \in \{x: \text{Bool}\} \]

(a) List all of the types that could replace the variable \( T \) if there are finitely many of them; otherwise write “Infinite”.

\( \text{Infinite: we can freely add fields, as long as there are at least the fields } x: \text{Bool} \text{ and } y \) (any type).

(b) What is the smallest type that \( T \) could be? If it doesn’t exist, write “None”.

\( \text{None} \)

(c) What is the largest type that \( T \) could be? If it doesn’t exist, write “None”.

\( \{x: \text{Bool}, y: \text{Top}\} \)

13.2

\[ \text{empty |- } \{x=(\forall b: \text{Top.} \ b) \ (b: \text{Top.} \ \text{true}), y=\text{true}, z=\text{unit}\} \in \{x: T\} \]

This question was actually not quite written the way we intended: instead of \( (\forall b: \text{Top.} \ b) \) we should have written \( (\forall b: \text{Top.} \ \text{true}) \). As stated, the correct answer is “None” to all three parts. Assuming (as many people did) that we had written what we intended to, the correct answers are as follows. (During grading we accepted these “morally correct” answers with a small penalty.)

(a) List all of the types that could replace the variable \( T \) if there are finitely many of them; otherwise write “Infinite”.

\( \text{Top, Bool} \)

(b) What is the smallest type that \( T \) could be? If it doesn’t exist, write “None”.

\( \text{Bool} \)

(c) What is the largest type that \( T \) could be? If it doesn’t exist, write “None”.

\( \text{Top} \)

13.3

\[ \text{empty |- } (\forall p: \{y: \text{Unit}, z: \text{Unit}\}. p.y) \in T \]

(a) What is the smallest type that \( T \) could be? If it doesn’t exist, write “None”.

\( \{y: \text{Unit}, z: \text{Unit}\} \rightarrow \text{Unit} \)

(b) What is the largest type that \( T \) could be? If it doesn’t exist, write “None”.

\( \text{Top} \)

13.4

\[ \text{empty |- } (\forall p: \text{Top.} \ p) \ (\forall z: \text{T.} \ \text{true}) \in \{x: \text{Top}, y: \text{Unit}\} \rightarrow \text{Bool} \]

In retrospect, we found this question confusing, so we gave full credit (4 points) to fully correct answers and 3 points to any other answer.

(a) What is the smallest type that \( T \) could be? If it doesn’t exist, write “None”.

\( \text{None} \) (the term does not have the claimed type, no matter what we put for \( T \))

(b) What is the largest type that \( T \) could be? If it doesn’t exist, write “None”.

\( \text{None} \)
Give a detailed informal proof of the following theorem about the subtype relation for the STLC with booleans and products (but no records) as defined on page 9. Please be sure to state your induction hypothesis explicitly at the beginning of each case where it is used.

**Theorem:** If $S_1 \rightarrow S_2 <: T$ then either $T = \text{Top}$ or $T = T_1 \rightarrow T_2$ for some $T_1$ and $T_2$ with $T_1 <: S_1$ and $S_2 <: T_2$.

You may assume the following lemma (“Nothing Above Top”):

If $\text{Top} <: T$, then $T = \text{Top}$.

**Proof:** By induction on the given subtyping derivation. To be very pedantic, we are proving the proposition

for all $S_1$ and $S_2$, if $S = S_1 \rightarrow S_2$ then either $T = \text{Top}$ or there are some $T_1$ and $T_2$ such that $T = T_1 \rightarrow T_2$ and $T_1 <: S_1$ and $S_2 <: T_2$,

by induction on a derivation of $S <: T$.

- **Case $S\_\text{Trans}$:** For some $U$, we have $S <: U$ and $U <: T$, plus two induction hypotheses:
  - (a) for all $S_1$ and $S_2$, if $S = S_1 \rightarrow S_2$ then either $U = \text{Top}$ or there are some $U_1$ and $U_2$ such that $U = U_1 \rightarrow U_2$ and $U_1 <: S_1$ and $S_2 <: U_2$;
  - (b) for all $U_1$ and $U_2$, if $U = U_1 \rightarrow U_2$ then either $T = \text{Top}$ or there are some $T_1$ and $T_2$ such that $T = T_1 \rightarrow T_2$ and $T_1 <: U_1$ and $U_2 <: T_2$.

Since we know $S = S_1 \rightarrow S_2$, the first induction hypothesis gives us two cases.

- If $U = \text{Top}$, we appeal to the “Nothing Above Top” Lemma to show that $T$ is $\text{Top}$.
- Otherwise, there are some $U_1$ and $U_2$ such that $U = U_1 \rightarrow U_2$ and $U_1 <: S_1$ and $S_2 <: U_2$. We apply the second induction hypothesis, yielding two more subcases.
  - * If $T = \text{Top}$, we are done.
  - * Otherwise, there are some $T_1$ and $T_2$ such that $T = T_1 \rightarrow T_2$ and $T_1 <: U_1$ and $U_2 <: T_2$. Since $U_1 <: S_1$ and $T_1 <: U_1$, we have $T_1 <: S_1$ by $S\_\text{Trans}$, and similarly $T_2 <: U_2$.

- **Case $S\_\text{Refl}$:** Then $S = T$ by the form of the rule. Since we know $S = S_1 \rightarrow S_2$, the fact that $T = S_1 \rightarrow S_2$ is immediate, while $S_1 <: S_1$ and $S_2 <: S_2$ follow by reflexivity.

- **Case $S\_\text{Top}$:** Then $T = \text{Top}$ by the form of the rule, and the result is immediate.

- **Case $S\_\text{Prod}$:** Then, by the form of the rule, $S$ must be a product and the result is trivial (since $S$ cannot be both a product and an arrow type). (The induction hypotheses are not used.)

- **Case $S\_\text{Arrow}$:** Then by the form of the rule, $T = T_1 \rightarrow T_2$ (and $S = S_1 \rightarrow S_2$, which we already knew), with $T_1 <: S_1$ and $S_2 <: T_2$, and the result is immediate. (The induction hypotheses are not used.)
For completeness, here is a condensed proof of the given lemma (that you did not have to prove).

**Lemma ["Nothing Above Top"]:** If \( \text{Top} <: T \), then \( T = \text{Top} \).

**Proof:** Straightforward induction on the given subtyping derivation. The cases for \( \text{S Prod} \) and \( \text{S Arrow} \) are impossible. The three remaining cases either are immediate from the form of the rule (\( \text{S Refl} \) and \( \text{S Top} \)) or follow directly from the induction hypotheses (\( \text{S Trans} \)). □
For Reference

Total maps

Definition total_map (A : Type) := string -> A.

Definition t_empty {A : Type} (v : A) : total_map A :=
  (fun _ => v).

Definition t_update {A : Type} (m : total_map A)
  (x : string) (v : A) :=
  (* eqb_string : string -> string -> bool *)
  fun x' => if eqb_string x x' then v else m x'.

Notation "x '!->' v ';' m" := (t_update m x v)
Notation "a '!->' x" := (t_update empty_st a x).

Useful facts about maps

Lemma t_apply_empty : forall (A : Type) (x : string) (v : A),
  (_ !-> v) x = v.

Lemma t_update_eq : forall (A : Type) (m : total_map A) x v,
  (x !-> v ; m) x = v.

Lemma t_update_neq : forall (A : Type) (m : total_map A) x1 x2 v,
  x1 <> x2 ->
  (x1 !-> v ; m) x2 = m x2.

Lemma t_update_shadow : forall (A : Type) (m : total_map A) x v1 v2,
  (x !-> v2 ; x !-> v1 ; m) = (x !-> v2 ; m).

Lemma t_update_same : forall (A : Type) (m : total_map A) x,
  (x !-> m x ; m) = m.

Lemma t_update_permute : forall (A : Type) (m : total_map A) v1 v2 x1 x2,
  x2 <> x1 ->
  (x1 !-> v1 ; x2 !-> v2 ; m)
  = (x2 !-> v2 ; x1 !-> v1 ; m).

1
Formal definitions for Imp

Syntax

Inductive aexp : Type :=
  | ANum : nat -> aexp
  | AId : string -> 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 : string -> aexp -> com
  | CSeq : com -> com -> com
  | CIF : bexp -> com -> com -> com
  | CWhile : bexp -> com -> com.

Infix ".+" := APlus (at level 50).
Infix "+." := AMinus (at level 50).
Infix ".*" := AMult (at level 40).
Infix "+." := BLe (at level 70).
Infix "+." := BEq (at level 70).
Infix "+.&&" := BAnd (at level 80).
Notation "!' b" := (BNot b) (at level 60).

Notation "+SKIP++" := CSkip.
Notation "x ':=' a" := (CAss x a).
Notation "c1 ;; c2" := (CSeq c1 c2).
Notation "'WHILE' b 'DO' c 'END''" := (CWhile b c).
Notation "'IFB' b 'THEN' c1 'ELSE' c2 'FI''" := (CIf b c1 c2).

There are also implicit coercions from nat and string to aexp, and from bool to bexp.
Evaluation functions for expressions

Definition state := total_map nat.

Fixpoint aeval (st : state) (a : aexp) : nat :=
    match a with
    | ANum n => n
    | AId x => st x
    | APlus a1 a2 => (aeval st a1) + (aeval st a2)
    | AMinus a1 a2 => (aeval st a1) - (aeval st a2)
    | AMult a1 a2 => (aeval st a1) * (aeval st a2)
    end.

Fixpoint beval (st : state) (b : bexp) : bool :=
    match b with
    | BTrue => true
    | BFalse => false
    | BEq a1 a2 => (aeval st a1) =? (aeval st a2)
    | BLe a1 a2 => (aeval st a1) <=? (aeval st a2)
    | BNot b1 => negb (beval st b1)
    | BAnd b1 b2 => andb (beval st b1) (beval st b2)
    end.
Evaluation relation for commands

Inductive ceval : com -> state -> state -> Prop :=
  | E_Skip : forall st,
    st = [ SKIP ] => st
  | E_Ass : forall st a1 n x,
    aeval st a1 = n ->
    st = [ x ::= a1 ] => (x !-> n ; st)
  | E_Seq : forall c1 c2 st st' st'',
    st = [ c1 ] => st' ->
    st' = [ c2 ] => st'' ->
    st = [ c1 ;; c2 ] => st''
  | E_IfTrue : forall st st' b c1 c2,
    beval st b = true ->
    st = [ c1 ] => st' ->
    st = [ IFB b THEN c1 ELSE c2 FI ] => st'
  | E_IfFalse : forall st st' b c1 c2,
    beval st b = false ->
    st = [ c2 ] => st' ->
    st = [ IFB b THEN c1 ELSE c2 FI ] => st'
  | E_WhileFalse : forall b st c,
    beval st b = false ->
    st = [ WHILE b DO c END ] => st
  | E_WhileTrue : forall st st'' b c,
    beval st b = true ->
    st = [ c ] => st' ->
    st' = [ WHILE b DO c END ] => st'' ->
    st = [ WHILE b DO c END ] => st''

where "st = [ c ] => st'" := (ceval c st st').

Program equivalence

Definition aequiv (a1 a2 : aexp) : Prop :=
  forall (st : state), aeval st a1 = aeval st a2.

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),
  (st = [ c1 ] => st') <-> (st = [ c2 ] => st').
Hoare Logic

Definition hoare_triple

\[(P : \text{Assertion}) (c : \text{com}) (Q : \text{Assertion}) : \text{Prop} := \forall st \text{ st}', \text{ st} = [c] \Rightarrow \text{ st}' \rightarrow P \text{ st} \rightarrow Q \text{ st}'.\]

Notation "\{\{ P \}\} c \{\{ Q \}\}" := (hoare_triple P c Q).

Definition assert_implies (P Q : \text{Assertion}) : \text{Prop} := \forall st, P \text{ st} \rightarrow Q \text{ st}.

Notation "P ->> Q" := (assert_implies P Q).

Hoare logic rules

------------------------------------ (hoare_asgn)
\{\{ \text{assn_sub x a Q} \}\} x ::= a \{\{ Q \}\}

-------------------- (hoare_skip)
\{\{ P \}\} \text{ SKIP} \{\{ P \}\}

\{\{ P \}\} c1 \{\{ Q \}\} \{\{ Q \}\} c2 \{\{ R \}\}

----------------------------------------- (hoare_seq)
\{\{ P \}\} c1;;c2 \{\{ R \}\}

\{\{ P \} /\ b \}\} c1 \{\{ Q \}\} \{\{ P \} /\ \neg b \}\} c2 \{\{ Q \}\}

----------------------------------------- (hoare_if)
\{\{ P \}\} \text{ IFB b THEN c1 ELSE c2 FI} \{\{ Q \}\}

\{\{ P \} /\ b \}\} c \{\{ P \}\}

------------------------- (hoare_while)
\{\{ P \}\} \text{ WHILE b DO c END} \{\{ P \} /\ \neg b \}\}

\text{P ->> P'} \{\{ P' \}\} c \{\{ Q' \}\} \text{ Q' ->> Q}

---------------------------------------- (hoare_consequence)
\{\{ P \}\} c \{\{ Q \}\}
STLC with booleans

Syntax

\[
\begin{align*}
T &::= \text{Bool} \\
& | T \rightarrow T \\
& | \lambda x : T. t \\\n& | \text{true} \\
& | \text{false} \\
& | \text{if } t \text{ then } t \text{ else } t
\end{align*}
\]

Small-step operational semantics

\[
\begin{align*}
\text{value } v \\
\text{---------------------- (ST_AppAbs)} \\
(\lambda x : T. t) \ v \rightarrow [x:=v]t
\end{align*}
\]

\[
\begin{align*}
t_1 \rightarrow t_1' \\
\text{------------------ (ST_App1)} \\
t_1 \ t_2 \rightarrow t_1' \ t_2
\end{align*}
\]

\[
\begin{align*}
\text{value } v_1 \\
\text{------------------ (ST_App2)} \\
v_1 \ t_2 \rightarrow v_1 \ t_2'
\end{align*}
\]

\[
\begin{align*}
\text{(if true then } t_1 \text{ else } t_2) \rightarrow t_1 \\
\text{------------------ (ST_IfTrue)}
\end{align*}
\]

\[
\begin{align*}
\text{(if false then } t_1 \text{ else } t_2) \rightarrow t_2 \\
\text{------------------ (ST_IfFalse)}
\end{align*}
\]

\[
\begin{align*}
t_1 \rightarrow t_1' \\
\text{------------------ (ST_If)} \\
(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) \rightarrow (\text{if } t_1' \text{ then } t_2 \text{ else } t_3)
\end{align*}
\]

Typing

\[
\begin{align*}
\Gamma \ x = T \\
\text{------------------ (T_Var)} \\
\Gamma, x : T_1 \vdash t_2 \in T_2
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash x \in T \\
\text{------------------ (T_Var)} \\
\Gamma, x : T_11 \vdash t_2 \in T_12
\end{align*}
\]

\[
\Gamma, x : T_11 \vdash t_2 \in T_11 \rightarrow T_12
\]

\[
\text{------------------ (T_App)}
\]

\[
\begin{align*}
\Gamma, x : T_11 \vdash t_2 \in T_11 \\
\Gamma, x : T_11 \vdash t_1 \ t_2 \in T_12 \\
\text{------------------ (T_App)} \\
\Gamma \vdash \text{true } \in \text{Bool}
\end{align*}
\]

\[
\Gamma \vdash \text{false } \in \text{Bool}
\]

(Continued on next page.)
\[ \begin{align*} \Gamma \vdash t_1 \in \text{Bool} & \quad \Gamma \vdash t_2 \in T \quad \Gamma \vdash t_3 \in T \quad \text{----------------------------------------------------------------------------- (T\_If)} \\ \Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \in T \end{align*} \]

Properties of STLC

Theorem preservation : \(\forall t, t', T,\)
\(\text{empty } \vdash t \in T -\rightarrow (t \rightarrow t') -\rightarrow \text{empty } \vdash t' \in T.\)

Theorem progress : \(\forall t, T,\)
\(\text{empty } \vdash t \in T -\rightarrow \text{value } t \lor \exists \, t', t \rightarrow t'.\)

STLC with let

\[
\begin{align*}
t & : = \ldots \\
& \mid \text{let } x=t \text{ in } t \quad \text{let-binding}
\end{align*}
\]

Small-step operational semantics (added to STLC rules)

\[
\begin{align*}
t_1 & \rightarrow t_1' \\
\text{let } x=t_1 \text{ in } t_2 & \rightarrow \text{let } x=t_1' \text{ in } t_2 \\
\text{let } x=v_1 \text{ in } t_2 & \rightarrow [x:=v_1]t_2
\end{align*}
\]

Typing (added to STLC rules)

\[
\begin{align*}
\Gamma \vdash t_1 \in T_1 \quad x \rightarrow \text{T}_1; \quad \Gamma \vdash t_2 \in T_2 \\
\text{----------------------------------------------------------------------------- (T\_Let)} \\
\Gamma \vdash \text{let } x=t_1 \text{ in } t_2 \in T_2
\end{align*}
\]

Formal definition of STLC with let in Coq, for reference in exercise Substitution as a relation

Inductive \(\text{tm} : \text{Type} :\)
\[
\begin{align*}
\mid \text{var} : \text{string } \rightarrow \text{tm} \\
\mid \text{app} : \text{tm } \rightarrow \text{tm } \rightarrow \text{tm} \\
\mid \text{abs} : \text{string } \rightarrow \text{ty } \rightarrow \text{tm } \rightarrow \text{tm} \\
\mid \text{tlet} : \text{string } \rightarrow \text{tm } \rightarrow \text{tm } \rightarrow \text{tm} \\
(* \text{i.e., } [\text{let } x=t_1 \text{ in } t_2] *)
\end{align*}
\]
STLC with products

Extend the STLC with product types, terms, projections, and pair values:

\[ T ::= \ldots \quad t ::= \ldots \quad v ::= \ldots \]
\[ | T * T \quad | (t, t) \quad | (v, v) \]
\[ | t.fst \quad | t.snd \]

Small-step operational semantics (added to STLC rules)

- \( t_1 \rightarrow t_1' \) \hspace{2cm} \( t_2 \rightarrow t_2' \)
  \[ (t_1, t_2) \rightarrow (t_1', t_2) \quad (v_1, t_2) \rightarrow (v_1', t_2) \]
  \( (ST\_Pair1) \quad (ST\_Pair2) \)

- \( t_1 \rightarrow t_1' \) \hspace{2cm} \( t_1.fst \rightarrow t_1'.fst \)
  \( (v_1, v_2).fst \rightarrow v_1 \)
  \( (ST\_Fst1) \quad (ST\_FstPair) \)

- \( t_1 \rightarrow t_1' \) \hspace{2cm} \( t_1.snd \rightarrow t_1'.snd \)
  \( (v_1, v_2).snd \rightarrow v_2 \)
  \( (ST\_Snd1) \quad (ST\_SndPair) \)

Typing (added to STLC rules)

- \( \Gamma \vdash t_1 \in T_1 \quad \Gamma \vdash t_2 \in T_2 \)
  \[ (\Gamma \vdash (t_1, t_2) \in T_1 * T_2) \]
  \( (T\_Pair) \)

- \( \Gamma \vdash t \in T_1 * T_2 \)
  \[ (\Gamma \vdash t.fst \in T_1) \quad (\Gamma \vdash t.snd \in T_2) \]
  \( (T\_Fst) \quad (T\_Snd) \)
STLC with fix

Extend the STLC from pages 6 to 8 with a fixed-point operator:

\[ t ::= \ldots \]
\[ \mid \text{fix } t \quad \text{fixed-point operator} \]

Small-step operational semantics (added to STLC rules)

\[ t_1 \rightarrow t_1' \]
\[ \quad \rightarrow \quad \text{(ST_Fix1)} \]
\[ \text{fix } t_1 \rightarrow \text{fix } t_1' \]
\[ \quad \rightarrow \quad \text{(ST_FixAbs)} \]
\[ \text{fix } (\forall x:T_1.t_2) \rightarrow [x:=\text{fix } (\forall x:T_1.t_2)] t_2 \]

Typing (added to STLC rules)

\[ \Gamma \vdash t_1 \in T_1 \rightarrow T_1 \]
\[ \quad \rightarrow \quad \text{(T_Fix)} \]
\[ \Gamma \vdash \text{fix } t_1 \in T_1 \]

STLC with Booleans, Products and Subtyping

Extend the language from pages 6 to 8 with the type Top (terms and values remain unchanged):

\[ T ::= \ldots \]
\[ \mid \text{Top} \]

Add these rules that characterize the subtyping relation:

\[ S <: U \quad U <: T \]
\[ \quad \rightarrow \quad \text{(S_Trans)} \]
\[ S <: T \quad T <: T \quad S <: \text{Top} \]
\[ \quad \rightarrow \quad \text{(S_Refl)} \quad \text{(S_Top)} \]
\[ S_1 <: T_1 \quad S_2 <: T_2 \]
\[ \quad \rightarrow \quad \text{(S_Prod)} \]
\[ S_1 * S_2 <: T_1 * T_2 \quad S_1 -> S_2 <: T_1 -> T_2 \]
\[ \quad \rightarrow \quad \text{(S_Arrow)} \]

And add this to the typing relation:

\[ \Gamma \vdash t \in S \quad S <: T \]
\[ \quad \rightarrow \quad \text{(T_Sub)} \]
\[ \Gamma \vdash t \in T \]
Record Subtyping

Implicitly, labels of a record should not contain any repetition.

\[
T ::= \ldots | \{i_1:T_1, \ldots, i_n:T_n\} | \{i_1=t_1, \ldots, i_n=t_n\} | \{i_1=v_1, \ldots, i_n=v_n\}
\]

\[
t ::= \ldots \quad v ::= \ldots \quad t.i
\]

Small-step operational semantics (added to STLC rules)

\[
t_i \rightarrow t_i' \quad \text{(ST.Rcd)}
\]

\[
\{i_1=v_1, \ldots, i_m=v_m, i_n=t_i, \ldots\} \rightarrow \{i_1=v_1, \ldots, i_m=v_m, i_n=t_i', \ldots\}
\]

Typing (added to STLC rules)

\[
\Gamma |- t_1 \in T_1 \ldots \Gamma |- t_n \in T_n \quad \text{(T.Rcd)}
\]

\[
\Gamma |- \{i_1=t_1, \ldots, i_n=t_n\} \in \{i_1:T_1, \ldots, i_n:T_n\}
\]

\[
\Gamma |- t \in \{\ldots, i:T_i, \ldots\} \quad \text{(T.Proj)}
\]

Subtyping (added to STLC rules)

Adding fields gives a subtype (S.RcdWidth), we can apply subtyping to each field (S.RcdDepth), subtyping can reorder fields (S.RcdPerm).

\[
n > m
\]

\[
\{i_1:T_1, \ldots, i_n:T_n\} <: \{i_1:T_1, \ldots, i_m:T_m\} \quad \text{(S.RcdWidth)}
\]

\[
S_1 <: T_1 \ldots S_n <: T_n \quad \text{(S.RcdDepth)}
\]

\[
\{i_1:S_1, \ldots, i_n:S_n\} <: \{i_1:T_1, \ldots, i_n:T_n\} \quad \text{(S.RcdPerm)}
\]

\[
\{i_1:S_1, \ldots, i_n:S_n\} \text{ is a permutation of } \{j_1:T_1, \ldots, j_n:T_n\}
\]