Before starting, be sure that you understand the course policy on Academic Integrity. Download code06.tar from the course website, which contains Standard ML files for a lambda-calculus interpreter.

1. Functions and Subtyping

Recall the operational semantics and type system for the Simply-Typed Lambda Calculus with constants, records, and subtyping:

\[ e ::= c \mid x \mid \lambda x : \tau. e \mid e \, e \mid \{l_1=e_1; \ldots; l_n=e_n\} \mid e \, i \]
\[ v ::= c \mid x \mid \lambda x : \tau. e \mid \{l_1=v_1; \ldots; l_n=v_n\} \]
\[ \tau ::= \text{int} \mid \tau \rightarrow \tau \mid \{l_1:\tau_1; \ldots; l_n:\tau_n\} \]
\[ \Gamma ::= \cdot \mid \Gamma, x : \tau \]

\[ e \to_{\text{cbv}} e' \]

### E-Select
\[ \frac{\{l_1=v_1; \ldots; l_n=v_n\}, l_i \to_{\text{cbv}} v_i}{e \to_{\text{cbv}} e'} \]

### E-Apply
\[ \frac{(\lambda x : \tau_a. e_b) \, v_a \to_{\text{cbv}} e_b[v_a/x]}{(\lambda x : \tau_a. e_b) \, v_a \to_{\text{cbv}} e_b[v_a/x]} \]

### E-Apply1
\[ \frac{e_f \to_{\text{cbv}} e'_f}{e_f \, e_a \to_{\text{cbv}} e'_f \, e_a} \]

### E-Apply2
\[ \frac{e_a \to_{\text{cbv}} e'_a}{v_f \, e_a \to_{\text{cbv}} v_f \, e'_a} \]

### T-Const
\[ \frac{\Gamma(x) = \tau}{\Gamma \vdash c : \text{int}} \]

### T-Var
\[ \frac{\Gamma \vdash x : \tau}{\Gamma \vdash x : \tau} \]

### T-Lam
\[ \frac{\Gamma, x : \tau_a \vdash e_b : \tau_r}{\Gamma \vdash \lambda x : \tau_a. e_b : \tau_a \rightarrow \tau_r} \]

### T-Apply
\[ \frac{\Gamma \vdash e_f : \tau_a \rightarrow \tau_r \quad \Gamma \vdash e_a : \tau_a}{\Gamma \vdash e_f \, e_a : \tau_r} \]

### T-Record
\[ \frac{\Gamma \vdash \{l_1=\tau_1; \ldots; l_n=\tau_n\}}{\Gamma \vdash \{l_i:\tau_i; \ldots; l_n:\tau_n\} : \{l_1:\tau_1; \ldots; l_n:\tau_n\}} \]

### T-Select
\[ \frac{\Gamma \vdash e : \tau \quad \tau' \leq \tau}{\Gamma \vdash e.l_i : \tau_i} \]

### T-Subsumption
\[ \frac{\Gamma \vdash e : \tau' \quad \tau' \leq \tau}{\Gamma \vdash e : \tau} \]

### S-Ref1
\[ \frac{\tau \leq \tau}{\tau \leq \tau} \]

### S-Trans
\[ \frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3} \]

### S-Arrow
\[ \frac{\tau_a \rightarrow \tau_r \quad \tau_r' \leq \tau_r}{\tau_a \rightarrow \tau_r'} \]

### S-Width
\[ \frac{\{l_1:\tau_1; \ldots; l_n:\tau_n; l_i:\tau\}}{\{l_1:\tau_1; \ldots; l_i:\tau_i; l_{i+1}:\tau_{i+1}; \ldots; l_n:\tau_n\}} \leq \{l_1:\tau_1; \ldots; l_n:\tau_n\}} \]

Note that functions have an type-annotated argument \((\lambda x : \tau_a. e_b)\) and the T-Lam rule requires the body expression to be well-typed with the argument having its annotated type.
(a) Consider the following alternative S-ARROW rule for function subtyping, which is *covariant* in both the argument type and the result type:

\[
\frac{\tau_a' \leq \tau_a \quad \tau_r' \leq \tau_r}{\tau_a' \to \tau_r' \leq \tau_a \to \tau_r}
\]

Show that this rule is *unsound*.

To show that a rule is unsound:

i. give an example program

ii. show that the program is well-typed (by giving the typing derivation) with the rule

iii. show that the program gets stuck (by giving the evaluation steps)

(*Hint: 1(a) is (probably) easier than 1(b).* )

(b) Consider the following alternative S-ARROW rule for function subtyping, which is *contravariant* in both the argument type and the result type:

\[
\frac{\tau_a \leq \tau_a' \quad \tau_r \leq \tau_r'}{\tau_a \to \tau_r \leq \tau_a' \to \tau_r'}
\]

Show that this rule is *unsound*.

(*Hint: 1(b) is (probably) harder than 1(a).* )
2. References and Subtyping

We now extend the language from Problem 1 with a mutable heap (as in Homework 5):

\[
\begin{align*}
H & ::= \text{Id}, \text{Hv} \rightarrow v \\
\Gamma & ::= \text{Id}, \text{Hv} : \tau
\end{align*}
\]

\[
\begin{array}{llll}
\text{H;e} & \rightarrow_{\text{cbv}} & H';e'
\end{array}
\]

\[
\begin{array}{llll}
\text{E-Apply} & H;e \rightarrow_{\text{cbv}} H';e' & \text{E-Apply1} & H;e_a \rightarrow_{\text{cbv}} H';e'_a \\
H;\{l_1=v_1;\cdots;l_n=v_n\}.I \rightarrow_{\text{cbv}} H;v_i & H;e_a \rightarrow_{\text{cbv}} H';e'_a & H;e_f \rightarrow_{\text{cbv}} H';e'_f & H;v_f \rightarrow_{\text{cbv}} H';v'_f
\end{array}
\]

\[
\begin{array}{llll}
\text{E-Select} & H;\{l_1=v_1;\cdots;l_n=v_n\}.I \rightarrow_{\text{cbv}} H;v_i & \text{E-Select1} & H;e_a \rightarrow_{\text{cbv}} H';e'_a \\
H;\{l_1=v_1;\cdots;l_n=v_n\}.I \rightarrow_{\text{cbv}} H;v_i & H;\{l_1=v_1;\cdots;l_n=v_n\}.I \rightarrow_{\text{cbv}} H';v_i & H;\{l_1=v_1;\cdots;l_n=v_n\}.I \rightarrow_{\text{cbv}} H';v_i
\end{array}
\]

\[
\begin{array}{llll}
\text{E-Alloc} & a \notin \text{Dom}(H) & \text{E-Alloc1} & H;\{l_1=v_1;\cdots;l_n=v_n\}.I \rightarrow_{\text{cbv}} H';e'_a \\
H;\text{ref } v_a \rightarrow_{\text{cbv}} H;v_a & H;\text{ref } e_a \rightarrow_{\text{cbv}} H';\text{ref } e'_a & H;\text{ref } e_a \rightarrow_{\text{cbv}} H';\text{ref } e'_a & H;\text{ref } e_a \rightarrow_{\text{cbv}} H';\text{ref } e'_a
\end{array}
\]

\[
\begin{array}{llll}
\text{E-Get} & H(a) = v & \text{E-Get1} & H;e_a \rightarrow_{\text{cbv}} H';e'_a \\
H;\{l_1=v_1;\cdots;l_n=v_n\}.I \rightarrow_{\text{cbv}} H;v & H;\{l_1=v_1;\cdots;l_n=v_n\}.I \rightarrow_{\text{cbv}} H';v & H;\{l_1=v_1;\cdots;l_n=v_n\}.I \rightarrow_{\text{cbv}} H';v & H;\{l_1=v_1;\cdots;l_n=v_n\}.I \rightarrow_{\text{cbv}} H';v
\end{array}
\]

\[
\begin{array}{llll}
\text{E-Set} & H[a] \leftarrow v = H' & \text{E-Set1} & H;e_r \rightarrow_{\text{cbv}} H';e'_r \\
H;\{l_1=v_1;\cdots;l_n=v_n\}.I \rightarrow_{\text{cbv}} H;v & H;e_r : e_a \rightarrow_{\text{cbv}} H';e'_r & H;e_r : e_a \rightarrow_{\text{cbv}} H';e'_r & H;e_r : e_a \rightarrow_{\text{cbv}} H';e'_r
\end{array}
\]

\[
\begin{array}{llll}
\Gamma \vdash c : \tau & \\
\text{T-Const} & & \text{T-Lam} & \\
\Gamma \vdash c : \text{int} & \Gamma \vdash x : \tau & \Gamma \vdash \lambda x: \tau, e_b : \tau \rightarrow \tau & \Gamma \vdash e_f : \tau_a \rightarrow \tau_v \\
\Gamma \vdash c : \tau & \Gamma \vdash x : \tau & \Gamma \vdash e_f : \tau_a \rightarrow \tau_v & \Gamma \vdash e_f : \tau_a \\
\text{T-Record} & & \text{T-Select} & \\
\Gamma \vdash e_1 : \tau_1 & \cdots & \Gamma \vdash e_n : \tau_n & \Gamma \vdash c : \{l_1: \tau_1; \cdots; l_n: \tau_n\} \\
\Gamma \vdash \{l_1=e_1; \cdots; l_n=e_n\} & \{l_1: \tau_1; \cdots; l_n: \tau_n\} & \Gamma \vdash e : \{l_1: \tau_1; \cdots; l_n: \tau_n\} & \Gamma \vdash e : \text{ref } \tau
\end{array}
\]

\[
\begin{array}{llll}
\text{T-Alloc} & & \text{T-Set} & \\
\Gamma \vdash e_a : \tau_a & \Gamma \vdash e_r : \text{ref } \tau_a & \Gamma \vdash e_r : \text{ref } \tau_a & \Gamma \vdash e_r : \text{ref } \tau_a \\
\Gamma \vdash \text{ref } e_a : \text{ref } \tau_a & \Gamma \vdash ! e_r : \tau_a & \Gamma \vdash e_r : \tau_a & \Gamma \vdash ! e_r : \text{ref } \tau_a
\end{array}
\]

\[
\begin{array}{llll}
\text{T-Addr} & \Gamma \vdash a : \text{ref } \tau & & \\
\Gamma \vdash a : \text{ref } \tau & \Gamma \vdash e_r : \tau_a & \Gamma \vdash e_r : \tau_a & \Gamma \vdash e_r : \tau_a \\
\Gamma \vdash e : \tau & \Gamma \vdash e : \tau & \Gamma \vdash e : \tau & \Gamma \vdash e : \tau
\end{array}
\]

\[
\begin{array}{llll}
\text{T-Subsumption} & \Gamma \vdash e : \tau' & \tau' \leq \tau & \\
\Gamma \vdash e : \tau & \Gamma \vdash e : \tau & \Gamma \vdash e : \tau & \Gamma \vdash e : \tau
\end{array}
\]

\[
\begin{array}{llll}
\text{S-Ref} & & \text{S-Perm} & \\
\tau \leq \tau & \tau \leq \tau & \tau \leq \tau & \text{ref } \tau \leq \text{ref } \tau
\end{array}
\]

\[
\begin{array}{llll}
\text{S-Trans} & \tau_1 \leq \tau_2 & \tau_2 \leq \tau_3 & \tau_3 \leq \tau_4 & \text{S-Subtype} & \tau_a \leq \tau'_a & \tau'_a \leq \tau_r & \tau_r \leq \tau_v & \text{S-Subtype} & \tau_a \leq \tau'_a & \tau'_a \leq \tau_r & \tau_r \leq \tau_v
\end{array}
\]

\[
\begin{array}{llll}
\text{S-Width} & \{l_1: \tau_1; \cdots; l_n: \tau_n; l: \tau\} & \leq \{l_1: \tau_1; \cdots; l_n: \tau_n\} & \leq \{l_1: \tau_1; \cdots; l_n: \tau_n\} & \{l_1: \tau_1; \cdots; l_n: \tau_n; l: \tau\} & \leq \{l_1: \tau_1; \cdots; l_n: \tau_n\}
\end{array}
\]

\[
\begin{array}{llll}
\text{S-Perm} & \tau' \leq \tau & \tau \leq \tau' & \text{S-Perm} & \text{ref } \tau' \leq \text{ref } \tau & \text{ref } \tau \leq \text{ref } \tau
\end{array}
\]
(a) Consider the following alternative S-Ref rule for reference subtyping, which is *covariant* in the contained type:

\[
\begin{align*}
\text{S-Ref} & \\
\tau' & \leq \tau \\
\text{ref} & \quad \tau' \leq \text{ref} \quad \tau
\end{align*}
\]

Show that this rule is *unsound.*

*(Hint: 2(a) is (probably) harder than 2(b).*

(b) Consider the following alternative S-Ref rule for reference subtyping, which is *contravariant* in the contained type:

\[
\begin{align*}
\text{S-Ref} & \\
\tau & \leq \tau' \\
\text{ref} & \quad \tau' \leq \text{ref} \quad \tau
\end{align*}
\]

Show that this rule is *unsound.*

*(Hint: 2(b) is (probably) easier than 2(a).*

(c) Consider the given *sound* S-Ref rule for reference subtyping, which is *invariant* in the contained type:

\[
\begin{align*}
\text{S-Ref} & \\
\tau' & \leq \tau & \tau \leq \tau' \\
\text{ref} & \quad \tau' \leq \text{ref} \quad \tau
\end{align*}
\]

Show that this rule is *not admissible.*

To show that a rule is not admissible:

i. give an example program

ii. show that the program is well-typed (by giving the typing derivation) with the rule

iii. argue that the program is not well-typed without the rule

Note that the language already has reflexive subtyping; therefore, we can already derive \( \tau \leq \tau \) for all \( \tau \).

*(Hint: Find a pair of types \( \tau_1 \) and \( \tau_2 \) such that \( \tau_1 \leq \tau_2 \) and \( \tau_2 \leq \tau_1 \), but \( \tau_1 \neq \tau_2 \).*

*(Hint: The simplest solution makes essential use of the type-annotation in \( \lambda x : \tau. \ e \).)*
3. Implementing Subtyping

The code provided defines an abstract syntax and a scanner/parser for the simply-typed lambda calculus with integers, addition, multiplication, greater-than, integer-based conditionals (0 is false, other integers are true), tuples, a heap, subtyping, and type aliases. (It is based closely upon the code from Homework 05.) Some important notes:

- A program begins with zero or more “type aliases” of the form `type s = τ;`, where `s` is an identifier and `τ` is a type. A type alias makes `s` a type name that can be used in other types. With respect for subtyping, both `s ≤ τ` and `τ ≤ s`. (The provided type checker ensures that a program’s type aliases have no cyclic references and that each type alias defines a different type name.)

- The type checker does not allow implicit subsumption. Instead, the syntax `(e : τ)` indicates an explicit subsumption: if `e` has the type `τ'` and `τ' ≤ τ`, then `(e : τ)` has the type `τ`. (If `τ'` is not a subtype of `τ`, then `(e : τ)` should not type check.)

- Tuple types are written `τ₁ * τ₂ * · · · * τₙ` and tuple expressions are written `e₁ * e₂ * · · · * eₙ`. (There is no syntax for tuple types or tuple expressions with fewer than two components, although the interpreter and typechecker support such tuple types and tuple expressions.)

- Tuple selections are written `e.i`, where `i` is an integer and the fields are numbered left-to-right starting with 1.

Implement the `subtype` function to support the following:

- The integer type is a subtype of the integer type.
- Function types have their usual contravariant argument and covariant result subtyping.
- Reference types are invariant (as described in Problem 2c).
- Tuple types have width and depth subtyping (but no permutation subtyping).
- A type name is a subtype of what it aliases and vice-versa

Use pattern matching on pairs of types. Also, using functions from the `ListPair` structure can make your solution more concise. The reference solution is only 15 lines long; ask for help if you find yourself attempting to write significantly more than this.
4. **Debriefing**
   - How many hours did you spend on this assignment?
   - Would you rate it as easy, moderate, or difficult?
   - How deeply do you feel you understand the material it covers (0% – 100%)?
   - If you have any other comments about the assignment, then please include them with your submission or send email to mtf@cs.rit.edu.

**Submission**

Programming components of the assignment:
- problem 3

must be submitted to the *Homework 6* Assignment on *MyCourses* by the due date.

Written components of the assignment:
- problems 1a.(i) (program), 1a.(ii) (typing derivation), 1a.(iii) (stuck evaluation)
- problems 1b.(i), 1b.(ii), 1b.(iii)
- problems 2a.(i), 2a.(ii), 2a.(iii)
- problems 2b.(i), 2b.(ii), 2b.(iii)
- problems 2c.(i) (program), 2c.(ii) (typing derivation), 2c.(iii) (argument)
- debriefing

must either be submitted to the instructor in class or submitted as a PDF file to the *Homework 6* Assignment on *MyCourses* by the due date.

**Document History**

**November 08, 2019**

Original version