Cornell University
Computer Science 6110

Homework 5 DUE: Tuesday, April 29, 11:59PM

Updates

- None yet.

What to turn in

Turn in the assignment by midnight on CMSX on the due date, including both code and written problems.

Working with a partner

You may have a partner on this assignment. Both partners are expected to work on all parts of the assignment, and to understand the solution.

1. **Strong normalization** [25 pts]

   (a) [20 pts]
   Show that all expressions in the language $\lambda^{\rightarrow +}$ are strongly normalizing by extending the proof of strong normalization for $\lambda^{\rightarrow}$. (This is the language where we extend the simply typed lambda calculus with sum types and the associated terms.)

   (b) [5 pts] Where does this style of proof fail if we add (iso)recursive types to the language? Discuss briefly. (This failure motivates the use of *step-indexed logical relations* for proving results about languages with similar features.)

2. **Explicit initialization** [35 pts]

   Compound data structures such as arrays, tuples, and records often need to be initialized step by step, rather than being created all at once.

   For example, in the Java language, when an object is created, before the execution of its constructor, all the non-primitive-typed fields have the default value null. The object is then gradually initialized using individual assignments to the fields. This is unsatisfactory because it means that null is a element of every type, and as a result, every object operation can potentially raise an exception.

   Now let us try to model safe, step-by-step initialization of tuples using an extension to the simply-typed λ-calculus:

$$
\begin{array}{rcl}
e & ::= & x \mid e_1\ e_2 \mid \lambda x{:}\tau.\, e \mid b \mid \mathsf{malloc}(\tau_1 * \ldots * \tau_n) \mid e[n] \mid e_1[n] := e_2 \\
b & ::= & 0 \mid 1 \mid 2 \mid \ldots \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{null} \\
\tau \in \textbf{Type} & ::= & B \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 * \ldots * \tau_n \mid (\tau_1 * \ldots * \tau_n) \backslash \{n_1, \ldots, n_k\} \\
B & ::= & \mathsf{int} \mid \mathsf{bool} \mid 1 \\
v \in \textbf{Value} & ::= & b \mid \lambda x{:}\tau.\, e \mid (v_1, \ldots, v_n)
\end{array}
$$

   In order to create a tuple, the expression $\mathsf{malloc}(\tau_1 * \ldots * \tau_n)$ is used, rather than $(e_1, \ldots, e_n)$, which creates a fully initialized tuple at once. The result of $\mathsf{malloc}(\tau_1 * \ldots * \tau_n)$ is a fully *uninitialized* tuple, $(\mathsf{null}, \ldots, \mathsf{null})$, with a *masked* type $(\tau_1 * \ldots * \tau_n) \backslash \{1, \ldots, n\}$.[1]

[1] For a more elaborate version of masked types, see http://www.cs.cornell.edu/Projects/jmask

The type $(\tau_1 * \ldots * \tau_n) \backslash \{n_1, \ldots, n_k\}$ represents a tuple that has not been fully initialized: the elements numbered $n_1, \ldots, n_k$ are *masked*, that is, not initialized and have the value null, and the tuple, after being fully initialized, should have the type $\tau_1 * \ldots * \tau_n$. For convenience, we assume $\tau_1 * \ldots * \tau_n$ is equivalent to $(\tau_1 * \ldots * \tau_n) \backslash \{\}$, that is, the mask is empty.

Tuples are initialized functionally with expressions $e_1\,[n]\ := e_2$, in which e_1 first evaluates to a tuple with its n-th element being null (therefore masked in the type of e_1), and e_2 evaluates to a value that is compatible with the type of the n-th element in the tuple. The expression will generate a new tuple with its n-th element initialized, and otherwise the same as the result of e_1.

For example, the following expression will evaluate to a pair $(1, 2)$ of type $\mathsf{int} * \mathsf{int}$.

$$\begin{array}{l}(\lambda x{:}\,(\mathsf{int} * \mathsf{int}) \backslash \{2\}.\, x\,\texttt{[2]} := 2) \\ \quad (\lambda x{:}\,(\mathsf{int} * \mathsf{int}) \backslash \{1, 2\}.\, x\,\texttt{[1]} := 1) \\ \qquad \mathsf{malloc}(\mathsf{int} * \mathsf{int})\end{array}$$

On the other hand, the following two expressions would not type-check:

- $(\lambda x{:}\,\mathsf{int}.\, x)((\mathsf{malloc}(\mathsf{int} * \mathsf{int})\,\texttt{[1]}) := 2)\,\texttt{[2]}$
  The second element of the tuple has not been initialized, and therefore accessing it cannot get a value of type int.
- $((\mathsf{malloc}(\mathsf{int} * \mathsf{int})\,\texttt{[1]}) := 2)\,\texttt{[1]} := 1$
  The first element of the tuple cannot be initialized twice.

Finally, although the source language does not include an expression for explicit tuples, we augment the grammar in order to define the small-step semantics.

$$e ::= \ldots \mid (v_1, \ldots, v_n)$$

(a) [10 pts] Extend the definition of the evaluation context and small-step operational semantics of the simply-typed λ-calculus to include the three new expressions: $\mathsf{malloc}(\tau_1 * \ldots * \tau_n)$, $e\,[n]$, and $e_1\,[n]\ := e_2$.

(b) [10 pts] Extend the static semantics for the new expressions.

(c) [25 pts] Now let us try to prove the soundness of the language:

i. Formulate and prove a preservation lemma for the language, which states that evaluation preserves the type of the expression.

ii. Formulate and prove a progress lemma, which states that a well-typed program is either in a normal form, or can be stepped into another program,

iii. Formulate the soundness theorem.

(d) [0 pts] BONUS QUESTION: Reinitialization.

The language, as described aboved, ensures that every element in a tuple is initialized *exactly* once. However, it is sometimes desirable to enforce *reinitialization*, for example, to disallow further accesses to some sensitive data, or when the computation is staged, to give the next stage a fresh start.

Update the operational and static semantics to support enforcement of reinitializing an already initialized element of a tuple, while still ensuring the soundness. You do not have to redo the soundness proof. (Hint: consider some subtyping relationship between $\tau_1 * \ldots * \tau_n$ and $(\tau_1 * \ldots * \tau_n) \backslash k$).

3. **Implementing type inference** [45 pts]

The file typecheck.ml contains a template of a type inference algorithm for the simply-typed lambda calculus. This algorithm comprises two main parts, which you will implement:

(a) **Unification**: The function unify implements Robinson's algorithm to find the weakest unifier of two types. It should return a substitution (list of bindings from type variables to types) that unifies the types, or raise a TypeError exception if they cannot be unified.

(b) **Type Inference**: The function infer infers the type of a given expression for a given typing context. It should return the inferred type and a substitution containing all the constraints that were generated during the inference process. The latter of these is useful when recursively applying inference on sub-expressions. If the expression cannot be typed, raise a TypeError exception.

We provide a number of helper functions to get you started, including those for applying and composing substitutions, as well as generating fresh type variables. Use them as an aid!

You can verify your implementation by applying the function typecheck on expressions of your choice in utop, which should output the type of whatever you input (or raise an exception).