# CS520 Programming Assignment 4

**Posted: 10 Nov 2006**
**Due: 11:59pm, 1 Dec 2006**

**Overview** The purposes of this assignment is to implement the type inference for $\lambda_t$ (defined in programming assignment 2), including constraint generation, unification, etc.

**Syntax of $\lambda_t$**

The only difference from the definition in assignment 2 is that type annotations are optional (because we are able to infer them if not provided). For instance, for lambda abstraction, both `lam` $(x : ty) => t$ and `lam` $(x) => t$ are legal. Similarly for `letrec`.

**Constraint Typing Rules of $\lambda_t$**

Here, only rules that are not presented in [Handout 18, 19, 20] are provided.

$$\frac{\Sigma(op) = (T_1, \ldots, T_n) \to T \quad \Gamma \vdash t_1 : T_1' \mid C_1 \quad \ldots \Gamma \vdash t_n : T_n' \mid C_n}{\Gamma \vdash op(t_1, \ldots, t_n) : T \mid C_1 \cup \ldots \cup C_n \cup \{T_1' \doteq T_1\} \cup \ldots \cup \{T_n' \doteq T_n\}} \ (\textbf{CT-op})$$

$$\frac{\Gamma \vdash t_0 : T_0 \mid C_0 \quad \Gamma \vdash t_1 : T_1 \mid C_1 \quad \Gamma \vdash t_2 : T_2 \mid C_2}{\Gamma \vdash \texttt{if}\ t_0\ \texttt{then}\ t_1\ \texttt{else}\ t_2 : T_1 \mid C_0 \cup C_1 \cup C_2 \cup \{T_0 \doteq \texttt{bool}\} \cup \{T_1 \doteq T_2\}} \ (\textbf{CT-if})$$

$$\frac{\Gamma \vdash \texttt{let}\ x = \texttt{fix}(\lambda x.t_1)\ \texttt{in}\ t_2 : T \mid C}{\Gamma \vdash \texttt{letrec}\ x = t_1\ \texttt{in}\ t_2 : T \mid C} \ (\textbf{CT-letrec})$$

$$\frac{\Gamma \vdash t : T_1 \mid C_1 \quad X \text{ is fresh}}{\Gamma \vdash \texttt{fix}(t) : X \mid C_1 \cup \{X \to X \doteq T_1\}} \ (\textbf{CT-fix})$$

$$\frac{\Gamma \vdash t_1 : T_1 \mid C_1 \quad \ldots \quad \Gamma \vdash t_n : T_n \mid C_n}{\Gamma \vdash (t_1, \ldots, t_n) : T_1 * \ldots * T_n \mid C_1 \cup \ldots \cup C_n} \ (\textbf{CT-tup})$$

$$\frac{\Gamma \vdash t : T_1 * \ldots * T_n \mid C}{\Gamma \vdash t.i : T_i \mid C} \ (\textbf{CT-proj})$$

$$\frac{\Gamma \vdash t : T_1 \mid C_1}{\Gamma \vdash (t : T) : T \mid C_1 \cup \{T_1 \doteq T\}} \ (\textbf{CT-asc})$$

Note that the rule (**CT-proj**) requires that the term being projected has to have a tuple type in the form of $T_1 * \ldots * T_n$ and $i$ must be in $\{1, \ldots, n\}$. This in turn requires binding occurrences of variables of tuple types have to be explicitly type-annotated.

**Dynamic Semantics of $\lambda_t$**

The dynamic semantics is identical to that defined in assignment 2.

**Abstract Syntax Definition of $\lambda_t$ in Ocaml**

```
type varrep = {name: int; link: stp option ref}

and stp =
    TpBase of string
  | TpFun of stp * stp
  | TpTup of stp list
  | TpVar of varrep (* type variable *)
  | TpUni of int (* universal type variable *)

(* Abstract Syntax Tree (AST) definitions *)
type ttm =
    TtmBool of bool
  | TtmInt of int
  | TtmStr of string
  | TtmVar of string
  | TtmIf of ttm * ttm * ttm
  | TtmOp of string * (ttm list)
  | TtmLam of string * stp option * ttm
  | TtmApp of ttm * ttm
  | TtmLet of string * ttm * ttm
  | TtmLetrec of string * stp option * ttm * ttm
  | TtmTup of ttm list
  | TtmPro of ttm * int
  | TtmFix of ttm
  | TtmAsc of ttm * stp
```

We extend `stp` with *type variables* and *universal type variables* (for implementing type scheme). Note that each type variable is essentially a record with two fields: the first one is an integer which serves as a unique identifier for the variable; the second one is a reference so that instances of the same type variable refer to one same location. With this implementation, you don't need to implement substitution in the unification. Once the content of a location is changed, every instance of the same type variable is able to see the chance. The universal type variables are represented as integers and are merely used as place holders.

**Problem 1 (80pts+20pts):** Based on the given constraint typing rules, implement a function called `typecheck` in Ocaml which infers the type of a given $\lambda_t$ term. The `typecheck` function should be assigned the following type in Ocaml:

`typecheck : ttm` $\rightarrow$ `stp option`

Note that for a well-typed term $t$ of type $T$, `typecheck`$(t)$ should return `Some`$(T)$; Otherwise, return `None`.

Specifically, you need to generates all the constraints then perform the unification to solve the constraints (you need to adapt the algorithm in the book to handle tuple types). There are three phases in this problem and you can get partial credits by implementing some of them:

1. (**70pts**) Implement the type inference without supporting *let-polymorphism*.
2. (**10pts + 20pts**) Extend your implementation in 1 so that it handles *let-polymorphism*. There are two ways you can implement it: one based on term substitution is simpler (use the rule (CT-LetPoly) in Handout 19); the other based on generalizing free type variables is considered harder and for which there are suggestions at the end of Chapter 22 in Pierce's book. You can implement either of them. However, implementing the *second* alternative correctly will get you 20 extra points. Please state clearly which one you use in your submission.
3. (**0pts**) Based on the given dynamic semantics, implement a function called `eval` in Ocaml which *evaluates closed well-typed* $\lambda_t$ *terms through the call-by-value strategy (you can adapt the small-step semantics to big-step semantics for efficiency).* The `eval` function should have type

   `eval : ttm` $\rightarrow$ `ttm`

   in Ocaml. Note that this one is almost identical to the one you implemented in assignment 2 so no extra points is worth.

**Implementation notes** A few files (in **prog4.tar.gz**) are provided to start the assignment. You need to provide the actual implementations of the above functions based on the given code. Once all the code are ready, type **make** under the directory. If no error reported, an executable file called **evaluator** will be produced. You can test your code by typing

```
./evaluator filename
```

where *filename* should be replaced by some actual file path. There are some test cases provided in the **test** directory.

**Grading** The grading of the assignment is based on whether the required functionalities are correctly implemented. Please make sure your code can be compiled and tested on **csa2** because all submissions will be tested on **csa2**. There are **10pts** for

1. if the code is well organized.

2. if errors are properly handled.
3. if the code has necessary comments.
4. etc.