# MP 12 – Type-checking explicitly-typed MiniOCaml

CS 421 – Spring 2013
Revision 1.0

**Assigned** Thursday, April 18, 2013
**Due** Tuesday, April 23, 09:30
**Extension** 48 hours (20% penalty)

## 1 Change Log

**1.0** Initial Release.

## 2 Objective

You will write a type-checker (not a type-inferencer) for an *explicitly*-typed version of OCaml. This is the algorithm that was discussed in lecture 23.

After completing this MP, you will have learned about the following concepts:

- Implementing type-checking for functional languages
- Polymorphic type systems

Our solution (that is, the part that you need to write) is about 60 lines of OCaml. The algorithm you are implementing is given in lecture 23 (mainly slides 19-21); the code we provide you is described in detail in this assignment.

## 3 Collaboration

Collaboration is NOT allowed in this assignment.

## 4 Style Requirements

As in all MPs, you will be expected to meet the following style requirements. Submissions that do not meet these requirements will not receive a grade until they are resubmitted with correct styling. (Note that these requirements will be checked manually by the grader and will not be enforced by `handin`, so acceptance by `handin` does not indicate correct style.)

- No long lines. Lines are 80 characters long.
- No tabs. Use spaces for indentation.
- Indents should be no more than 4 spaces, and must be used consistently.

We will only enforce the rules just listed, but a more comprehensive style guide can be found at `http://caml.inria.fr/resources/doc/guides/guidelines.en.html`.

# 5 What to submit

You will submit `mp12.ml` using the `handin` program. Copy `mp12-skeleton.ml` to `mp12.ml` and start working from there.

As in previous, once you have finished appropriate sections of your interpreter, you can use the `run` and `run_with_args` functions to test individual programs, or simply add programs to the test suite.

- Download `mp12grader.tar.gz`. This tarball contains all the files you need, including the common AST file (`mp12common.ml`), the MiniOCaml lexer and parser (`miniocamllex.mll` and `miniocamlparse.mly`), and the skeleton solution file (`mp12-skeleton.ml`).
- As always, once you extract the tarball, copy `mp12-skeleton.ml` to `mp12.ml` and start modifying the file. You will modify only the `mp12.ml` file, and it is the only file that will be submitted.
- Compile your solution with `make`. Run the `./grader` to see how well you do.
- **Make sure to add several more test cases to the `tests` file; follow the pattern of the existing cases to add a new case.**
- You may use the included `testing.ml` file to run tests interactively. Open the OCaml repl and type `#use "testing.ml";;` to load all of the related modules and enable testing. Further specifics on interactive testing are included in that file.

# 6 Syntax and Semantics

This version of MiniOCaml is even simpler than the one in MPs 9 and 10, in that the syntax is much more restricted. The concrete syntax is shown in Figure 1 (and in `miniocamlparse.mly` in the MP12 tarball). It is also *different*, in that it includes explicit types for variables bound in function definitions and let expressions. Here is an example:

```
let f:(alpha->alpha) = fun x:alpha -> x
in f[(int->int)->(int->int)] f[int->int]
```

More examples are given in the notes for lecture 23.

The main function you will write is `tcheck`, which takes an expression and a type environment and returns a type, or raises an exception.

# 7 The set-up

The idea of explicitly-typed OCaml, and the type-checking algorithm, are covered in Lecture 23. We will provide a parser and translation to abstract syntax. In addition, the skeleton file defines types like `typeterm` and `typeenv`, plus a number of useful functions. Your job is to write `tcheck` (plus a few auxiliary functions).

The abstract syntax of expressions and types is shown in Figure 2. Pretty much everything that does not add anything interesting concerning type-checking, like `If`, has been removed.

A type environment is a function from program variables to type schemes (i.e. types with some variable quantified, e.g. $\forall\alpha.\alpha$ `list` $\rightarrow \alpha$); we have chosen to use a "structural" implementation (i.e. we have chosen to represent an environment as a list). A type scheme is a pair: a type and a list of the names of the bound type variables. We have provided a function `freevars` that returns a list of the *non*-quantified variables occurring in a type scheme (i.e. the variables obtained by applying `getvars`, minus the bound variables).

This leaves you with four functions to define: `instanceOf` (check whether a given type is an instance of a given type scheme); `generalize` (create a type scheme from a type by quantifying some of its variables); `opTypes` (check the application of built-in operations like `Plus`); and finally `tcheck` (find the type of an expression in a type environment, or raise an exception). Details are given below.

# 8 Problems

We first describe the code we have provided in the skeleton file. It should be noted that this problem is much simpler than doing type inference, even type inference for the monomorphic type system; the total size of our solution is under 100 lines of code, and half of those are given to you.

The abstract syntax for terms and types is shown in Figure 2.

Aside from the abstract syntax, the most important definitions given in the skeleton are these:

```
type typeterm = BoolType | IntType | FunType of typeterm * typeterm | Typevar of id
 type typeenv = (string * typescheme) list
 and typescheme = typeterm * (string list)
```

As noted above, a type environment is a map from program variables to type schemes, which we have represented in the familiar way. A typescheme is a type in which some of the variables are quantified. Here, a typescheme ($\tau$, [$\alpha$;. . .; $\beta$]) represents the type $\forall\alpha, \ldots, \beta.\tau$.

The functions we provide are these (they are described in more detail in the code itself): `getvars` returns all the variables in a type term; `empty_te` is the empty type environment; `lookup` and `extend` are the basic fetch and add operations on type environments (note that lookup raises a `Not_found` exception if the variable is not there). `freevars` returns the variables free in a type *scheme* (so, if the type scheme is ($\tau$, *boundvars*), this is `getvars`($\tau$) with the variables in *boundvars* removed). Beyond that, `string_of_typeterm` and `string_of_typescheme` provide printable versions of types and typeschemes for debugging, and that's about it.

These are the functions you need to write:

```
let instanceOf (tau':typeterm) ((tau,bndvars):typescheme) : bool
let generalize (tau:typeterm) (env:typeenv) : typescheme
let opTypes (bop:binary_operation) (tau1:typeterm) (tau2:typeterm) : typeterm
let tcheck (t:exp) (gamma:typeenv) : typeterm
```

`instanceOf tau' ts` implements the "$\leq$" operation between types and type schemes. Basically, it is a simultaneous traversal of `tau` and `tau'`; they should match everywhere except possibly in places where `ts` has a type variable (which *must* be among its bound type variables). The only thing that makes this tricky is that you have to be sure that everywhere `tau'` has substituted a type in place of a type variable, it has substituted the exact same type. (In our implementation, we just kept track of every substitution as we traversed both trees, and then checked in the end to make sure all substitutions for any one type variable were the same; this is not very efficient, but it would be acceptable in your solution.) Example: `int list` $\rightarrow$ `int` and $\alpha$ `list` $\rightarrow \alpha$ are instances of $\forall\alpha.\alpha$ `list` $\rightarrow \alpha$, but `bool list` $\rightarrow$ `int` and `int list` $\rightarrow \alpha$ are not. See the last bullet point below for some more information about instantiation.

`generalize tau env` returns a type scheme whose typeterm is `tau` and whose bound variables are the variables in `tau` that are not free in `env`. See the first bullet point below for a bit more explanation, and pages 20 and 21 of lecture 23 for an example of why we should **not** bind ALL the type variables in `tau`.

`opTypes` is a simple function that implements the type-checking for the few binary operations we have included. It is called from `tcheck`. Given a binary operation `bop`, `opTypes` checks that the types of its arguments, `tau1` and `tau2`, are correct for the given binary operation. If so, it returns the return type of that operation; otherwise, it raises a "misapplied binary operation" error. Here are all the operators and their types:

LessThan : *int* $\rightarrow$ *int* $\rightarrow$ *bool*
GreaterThan : *int* $\rightarrow$ *int* $\rightarrow$ *bool*
And : *bool* $\rightarrow$ *bool* $\rightarrow$ *bool*
Or : *bool* $\rightarrow$ *bool* $\rightarrow$ *bool*
Plus : *int* $\rightarrow$ *int* $\rightarrow$ *int*
Minus : *int* $\rightarrow$ *int* $\rightarrow$ *int*
Div : *int* $\rightarrow$ *int* $\rightarrow$ *int*
Mult : *int* $\rightarrow$ *int* $\rightarrow$ *int*

`tcheck` is the main function you need to define. It returns the type of its argument in the given environment, or raises a `TypeError` exception. The algorithm is given in the notes of lecture 23, slide 19.

*typeterm* → `int` | `bool` | `(` *typeterm* `->` *typeterm* `)` | *id*

*exp* → *int* | `true` | `false` | *id* | *exp exp* | `fun` *id* : *typeterm* `->` *exp*
| `let` *id* : *typeterm* = *exp* `in` *exp* | *id* `[`*typeterm*`]`

Figure 1: Explicitly-typed MiniOcaml concrete syntax

```
type exp =
    Operation of exp * binary_operation * exp
  | Var of id
  | PolyVar of id * typeterm
  | IntConst of int
  | True | False
  | Let of id * typeterm * exp * exp
  | Fun of id * typeterm * exp
  | App of exp * exp

and typeterm =
    BoolType
  | IntType
  | FunType of typeterm * typeterm
  | Typevar of id

and binary_operation = LessThan | GreaterThan
  | And | Or
  | Plus | Minus | Div | Mult

and id = string;;
```

Figure 2: Explicitly-typed MiniOcaml abstract syntax

We want to point out a couple of aspects of this assignment that may be confusing:

- In type environments, some types are generalized (namely, let-bound variables), and some aren't (fun-bound variables). All are treated as type schemes, except that for the non-generalized types, the list of bound variables is empty. This can also be true of let-bound variables, if their types have no variables, or if all the variables in them are non-generalizable (see the precise definition of generalize in lecture 23 (slide 21) and in the comments in mp12-skeleton).
- In the explicitly-typed language, some variables have associated types and some don't. Any variable whose type scheme has generalized variables (i.e. whose bound variable list is non-empty) must appear with an instantiation of that type scheme; part of the job of type-checking is making sure that this is a valid instance of the type scheme. If a variable's type scheme does not have any bound variables — meaning either that it is a fun-bound variable or a let-bound variable without generalizable type variables — then it does not have to include the type instantiation. In that case, its type must *exactly match* the type given in the type environment, even including identical type variable names. In the abstract syntax, these two kinds of variable references are represented by `PolyVar` and `Var` constructors.