
MP 5 – A Unification-Based Type Inferencer

CS 421 – Spring 2007

Revision 1.0

Assigned March 8, 2007

Due March 15, 2007 23:59

Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

2 Objectives

Your objectives are:

- Become comfortable using record types and variant types, particularly as used in giving Abstract Syntax Trees.
- Become comfortable with the notation for semantic specifications.
- Understand the type-inference algorithm.

3 Background

One of the major objectives of this course is to provide you with the skills necessary to implement a language. There are three major components to a language implementation: the parser, the internal representation, and the evaluator. In this MP you will work on the middle piece, the internal representation.

An interpreter or a compiler represents an expression in a language with an *Abstract Syntax Tree* (AST), usually implemented by means of a user-defined type. Functions can be written that use this type to perform evaluations, preprocessing, . . . anything that can or should be done with a language. In this MP, you will write some functions that perform type inferencing using unification. This type-inferencer will appear again as a component in several future MPs.

3.1 Type Inferencing

The pattern for type inferencing is similar to the procedure used to verify an expression has a type. The catch is that you are not told the type ahead of time, you have to figure it out as you go. The procedure is as follows:

1. Infer the types of all the subexpressions. For each subexpression, you will get back a proof tree and a list of constraints.
2. Create a new proof tree from the subexpressions.
3. Create a new set of constraints by taking the union of the constraints of the subexpressions. Add any new constraints to this.
4. Return the new proof tree and new set of constraints.

In a separate phase we apply the set of constraints to the proof to finish inferring a type.

4 MicroML

Our language, MicroML, is based off of Standard ML (SML), which, like OCaml, is a variant of ML. Like OCaml, MicroML has two basic syntactic categories: the *declaration* and the *expression*. MicroML also has a top-level, where one enters in a sequence of declarations (which can be viewed as one giant declaration) to be evaluated.

4.1 Declarations

Remember that in OCaml, one writes a bunch of declarations at the top level, of the form `let x = e`. One does the same thing in MicroML, except that a declaration in MicroML looks like this:

```
val x = e
```

where x is a variable and e is a MicroML expression. As in OCaml, one may string a bunch of declarations together, like so:

```
val x = e1
val y = e2
val z = e3...
```

Just as a sequence of OCaml expressions separated by ';' is itself an expression, such a sequence of declarations in MicroML is itself a declaration.

4.2 Expressions

Expressions in MicroML are almost identical to expressions in OCaml. One difference is in the handling local declarations. Local declarations in MicroML look like this:

```
let d in e end
```

Here d is an arbitrary declaration, which as explained above can be a single declaration, a sequence of declarations, or a declaration containing a local declaration (see below).

For example, the following is legal MicroML code:

```
val res =
  let val x = 3 val y = 4
  in
    x + y
```

At top level, we have a declaration of the form `val res = e`. e has the form `let d = e'`, where d is itself a declaration (`val x = 3 val y = 4`).

4.3 Local Declarations

In OCaml, one has the ability to declare a variable local to an expression, e.g. with `let x = 3 in x`. Here `x = 3` is a declaration local to the expression `x`. Note that one can do this in MicroML using `let val x = 3 in x end`.

But MicroML is even more powerful: it allows us to make a declaration local to another declaration. For example:

```
local val x = 3 val y = 4
in
  val xpy = x + y
  val xty = x * y
```

The only way to do this in OCaml is as follows:

```
let (xpy, xty) =
  let x, y = 3, 4
  in
    (x + y, x * y)
```

One can see that in the second example, the identifiers `xpy` and `xty` are very far away from the values they are assigned to. One can see that local declarations are a desirable feature.

5 Given Code

You are given some initial code in a file `mp5common.ml`. It contains, among other things, an OCaml representation of a simple functional programming language called MicroML.

Types These are elements of type `expType` from the previous MP:

```
type expType = TyVar of int | TyConst of (string * expType list)
```

When inferring types, you will need to generate fresh type-variable names. For this, you may use the side-effecting function `fresh` that takes a unit and returns a fresh type variable. The index stored by `fresh` (initially set to 0) will keep on growing as you use `fresh`.

Expressions and Declarations

As mentioned above, an interpreter represents an expression using an AST. We define the types `exp` and `dec` to represent the (abstract forms of) expressions and declarations of MicroML.

```
type dec =
  Valbind of (string * exp)
  | ValbindRec of (string * exp)
  | Local of (dec * dec)
  | Seq of (dec * dec)
and exp =
  Bool of bool
  | Char of char
  | Int of int
  | Real of float
  | String of string
  | Pair of exp * exp
  | Id of string
  | LetIn of dec * exp
  | App of exp * exp
  | Fn of string * exp
  | IfThenElse of exp * exp * exp
```

There are companion functions `print_exp` and `print_dec` that print expressions and declarations in more readable forms.

Environments

We need an environment to store the types of the variables. An environment is a list of pairs mapping variables (strings) to values (types):

```
type env = (string * expType) list
```

One interacts with environments using the following functions, pre-defined in `mp5common.ml`:

```
(*environment operations*)
let make_env x y = ... (*create env with single pair*)
let rec lookup_env gamma x = ... (*look up x in gamma*)
let sum_env delta gamma = ... (*update gamma with all mappings in delta*)
let ins_env gamma x y = ... (*insert x->y into gamma*)

val make_env : string -> expType -> env = <fun>
val lookup_env : env -> string -> expType = <fun>
val sum_env : env -> env -> env = <fun>
val ins_env : env -> string -> expType -> env = <fun>
```

Type Judgments

From the lectures, you know that an *expression* type judgment has the form $\Gamma \vdash e : \tau$. This says that in the environment Γ , the expression e has type τ .

For MicroML, we will find that we also need *declaration* type judgments. Such a judgment has the form $\Gamma \vdash d \rightarrow \Delta$. This says that the declaration d updates Γ by adding in the declarations in Δ .

These concepts are represented by the following data structures:

```
type judgment_exp = { gamma:env; exp:exp; expType:expType }
type judgment_dec = { gamma1:env; dec:dec; delta2:env }
```

The pre-defined functions `print_jexp` and `print_dexp` print readable forms of these typing judgments.

Proofs

A proof is a judgment and a list of assumptions. The assumptions are themselves proofs. To represent this we have a type `proof`.

There are two types of proofs, a proof of a type judgment, and a proof of a declaration judgment:

```
type proof = ExpProof of judgment_exp * proof list
          | DecProof of judgment_dec * proof list
```

These proofs are just like the proofs you have been doing in class: remember that in class the initial judgment is written on the bottom of the page. A line is drawn on top of it, assumptions are written down, and then these assumptions are proved.

In our setting, since we are doing type inference and not just type judgment, we need to keep track of constraints as we build our proof. On your homework you were able to magically guess what the type of an expression was before you did the actual judgment, and so you never really had to do type inference.

Instead of just writing down a type judgment, we write something of the form, $\Gamma \vdash e : \tau \mid C$, where C is a set of constraints. An example:

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2 \quad \Gamma \vdash e_3 : \tau_3 \mid C_3}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau \mid \{\tau_1 = \text{bool}, \tau = \tau_2, \tau = \tau_3\} \cup C_1 \cup C_2 \cup C_3}$$

This is the if-then-else rule. Notice that when one recurses on the proofs for the types of e_1 , e_2 and e_3 , one collects sets of constraints each time. At the end of the proof, one unions together these three constraint sets, along with the constraints that $\tau_1 = \text{bool}$, $\tau = \tau_2$, and $\tau = \tau_3$.

These constraints will later be fed into the unification algorithm to determine the actual types of all types variables used in your proof.

Pervasives environment

There are some built-in operators (ids) whose types are pre-defined wherever they occur. Given the name `x` of one such operator, one can call `fresh_type (lookup_env pervasives x)` to retrieve the (possibly polymorphic) type of that operator.

The function `fresh_type` instantiates a polymorphic type with fresh variables.

```
let pervasives =
  let basic_type s = TyConst(s, [])
  in let unaryfn_type s = TyConst("->",[basic_type s;basic_type s])
  in let binaryfn_type s =
    TyConst("->",[TyConst("prod",[basic_type s;basic_type s]);
    basic_type s])
  in
  [
    ("true", basic_type "bool");
    ("false", basic_type "bool");
    ("and", binaryfn_type "bool");
    ("or", binaryfn_type "bool");
    ("not", unaryfn_type "bool");
    ("+", binaryfn_type "int");
    ("-", binaryfn_type "int");
    ("*", binaryfn_type "int");
    ("/", binaryfn_type "int");
    ("+.", binaryfn_type "real");
    ("-.", binaryfn_type "real");
    ("*.", binaryfn_type "real");
    ("**", binaryfn_type "real");
    ("nil", TyConst("list",[TyVar 0]));
    ("::", let tyList = TyConst("list",[TyVar 0]) in
      TyConst("->",[TyConst("prod",[TyVar 0;tyList]);tyList]));
    ("head", TyConst("->",[TyConst("list",[TyVar 0]);TyVar 0]));
    ("tail", TyConst("->",[TyConst("list",[TyVar 0]);TyVar 0]));
    ("=", TyConst("->",[TyConst("prod",[TyVar 0;TyVar 0]);basic_type "bool"]))
  ]
```

Note: code that uses pervasives is already given to you; you will never need to call these functions yourself.

Important pre-defined functions

Some important functions are pre-defined: The function `infer` takes in an `exp` and returns an `(expType * proof) option`. The first part is the type of the entire expression, and the second part is a proof (assuming success).

The functions `get_proof` and `get_ty` extract the proof and type parts, respectively (or raise an exception on `None`).

`print_proof` prints a proof in a tree-like form, with the root at the top, upside-down from the way we are used to seeing proofs.

```
# infer;;
- : exp -> (expType * proof) option = <fun>
# get_proof;;
- : ('a * 'b) option -> 'b = <fun>
# get_ty;;
```

```

- : ('a * 'b) option -> 'a = <fun>
# print_proof;;
- : proof -> unit = <fun>

```

You will see these functions used in examples below.

5.1 Your task

The body of the main type inferencing function, `infer`, is already implemented. `infer` takes an expression and returns (an option of) the type of the expression and a proof.

```

# infer;;
- : exp -> (expType * proof) option = <fun>

```

Your task is to finish the implementations of two functions needed by `infer`:

`gather_exp` takes in an expression and returns `None` (on failure), or some pair of a generic proof tree containing polymorphic variables, and a set of constraints to be unified.

`gather_dec` takes in an environment and a declaration and returns `None` (on failure), or some “update” environment, a proof, and a set of constraints.

6 Typing Expressions

Finish the implementation of `gather_exp` judgment, which takes in a judgment and returns `None` (on failure) or some pair whose first element is a proof, and whose second argument is a set of constraints to be unified.

```

# gather_exp;;
- : judgment_exp -> (proof * (expType * expType) list) option = <fun>

```

In any given rule, the type τ is the type passed in via the `judgment` argument to `gather_exp`, and any other τ_i or τ' are fresh variables.

6.1 Already Implemented

The if-then-else rule is already implemented. Look at it for inspiration.

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2 \quad \Gamma \vdash e_3 : \tau_3 \mid C_3}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau \mid \{\tau_1 = \text{bool}, \tau = \tau_2, \tau = \tau_3\} \cup C_1 \cup C_2 \cup C_3}$$

Note: The example below will work once you have special constants working.

```

# let tree12 = IfThenElse(Bool true, Int 1, Int 0);;
val tree12 : exp = IfThenElse (Bool true, Int 1, Int 0)
# print_proof (get_proof (infer tree12));;

```

```

{} |= if (true) then (1) else (0) : int
|--{} |= true : bool
|--{} |= 1 : int
|--{} |= 0 : int

```

```

- : unit = ()

```

6.2 Problems

1. (5 pts) Implement the special constants rule:

$$\frac{}{\Gamma \vdash c : \tau \mid \{\tau = \tau'\}} \text{ where } c \text{ is a special constant, and } \tau' \text{ is the corresponding primitive type}$$

Examples: `true` has type `Bool`, `'a'` has type `char`, `3` has type `int`, `3.4` has type `real`, and `"hi"` has type `string`.

Note: You will need to implement all 6 instances of this rule, one for each possible value of τ' .

Note: Type τ is passed in from as the “judgment” argument to `gather_exp`.

Note: The if-then-else rule is already implemented. Look at it for inspiration.

Note: Below, we name expressions `treei` in reference to abstract syntax trees (ASTs).

```
# let (tree1,tree2) = (Bool true,Int 3);;
val tree1 : exp = Bool true
val tree2 : exp = Int 3
# print_proof (get_proof (infer tree1));;

  |= true : bool

- : unit = ()
# print_proof (get_proof (infer tree2));;

  |= 3 : int

- : unit = ()
```

2. (5 pts) Implement the ID rule:

$$\frac{}{\Gamma \vdash id : \tau \mid \{\tau = \tau'\}} \text{ where } \Gamma(id) = \tau'$$

Note that $\Gamma(id)$ represents looking up the value of `id` in Γ . In OCaml, one writes `Mp5common.lookup_env gamma id`.

Note: Code is already in-place for looking up built-in operators.

```
# let tree6,tree7 = Id "x", Id "+";;
val tree6 : exp = Id "x"
val tree7 : exp = Id "+"
# infer tree6;;
- : (expType * proof) option = None
# print_proof (get_proof (infer tree7));;

  {} |= + : (int * int) -> int

- : unit = ()
```

3. (10 pts) Implement the pair rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2}{\Gamma \vdash (e_1, e_2) : \tau \mid \{\tau = \tau_1 * \tau_2\} \cup C_1 \cup C_2}$$

Note: Type τ is passed in from as the “judgment” argument to `gather_exp`. τ_1 and τ_2 are fresh variables.

```
# let tree8 = Pair(Int 3, String "hello");;
val tree8 : exp = Pair (Int 3, String "hello")
# print_proof (get_proof (infer tree8));;
```

```
{ } |= (3, "hello") : (int * string)
|--{ } |= 3 : int
|--{ } |= "hello" : string
```

```
- : unit = ()
```

4. (10 pts) Implement the let-in rule:

$$\frac{\Gamma \vdash d \Rightarrow \Delta \mid C_1 \quad \Delta + \Gamma \vdash e : \tau' \mid C_2}{\Gamma \vdash \text{let } d \text{ in } e : \tau \mid \{\tau = \tau'\} \cup C_1 \cup C_2}$$

Note: Declarations are handled in the next section, but the example given below should work since the code for `Valbind` is already given.

```
# let tree9 = LetIn(Valbind("x", Int 3), Id "x");;
val tree9 : exp = LetIn (Valbind ("x", Int 3), Id "x")
# print_proof (get_proof (infer tree9));;
```

```
{ } |= let (val x = 3) in (x) : int
|--{ } |= val x = 3 -> x->int
| |--{ } |= 3 : int
|--{x->int} |= x : int
```

```
- : unit = ()
```

5. (10 pts) Implement the application rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2}{\Gamma \vdash e_1 e_2 : \tau \mid \{\tau_1 = \tau_2 \rightarrow \tau\} \cup C_1 \cup C_2}$$

```
# let tree18 = App(Id "+", Pair(Int 3, Int 5));;
val tree18 : exp = App (Id "+", Pair (Int 3, Int 5))
# print_proof (get_proof (infer tree18));;
```

```
{ } |= (+)((3,5)) : int
|--{ } |= + : (int * int) -> int
|--{ } |= (3,5) : (int * int)
```

```

|--{} |= 3 : int
|--{} |= 5 : int

- : unit = ()
# let tree13 = App(Id "+",Pair(Int 3, App(Id "-",Pair(Int 10, Int 5))));;
...
# print_proof (get_proof (infer tree13));;

{} |= (+)((3,(-)((10,5)))) : int
|--{} |= + : (int * int) -> int
|--{} |= (3,(-)((10,5))) : (int * int)
|--{} |= 3 : int
|--{} |= (-)((10,5)) : int
|--{} |= - : (int * int) -> int
|--{} |= (10,5) : (int * int)
|--{} |= 10 : int
|--{} |= 5 : int

- : unit = ()

```

6. (10 pts) Implement the function rule:

$$\frac{\{x \rightarrow \tau_1\} + \Gamma \vdash e : \tau_2 \mid C}{\Gamma \vdash \text{fn } x \Rightarrow e : \tau \mid \{\tau = \tau_1 \rightarrow \tau_2\} \cup C}$$

```

# let tree10 = Fn("x", Id "x");;
val tree10 : exp = Fn ("x", Id "x")
# print_proof (get_proof (infer tree10));;

{} |= (fn x => x) : 23 -> 23
|--{x->23} |= x : 23

- : unit = ()
# let tree11 = App(Fn("x", Id "x"),Int 3);;
val tree11 : exp = App (Fn ("x", Id "x"), Int 3)
# print_proof (get_proof (infer tree11));;

{} |= ((fn x => x))(3) : int
|--{} |= (fn x => x) : int -> int
| |--{x->int} |= x : int
|--{} |= 3 : int

- : unit = ()

```

7 Typing Declarations

Finish the implementation `gather_dec gamma dec`, which takes in an environment Γ and a declaration, and returns `None` (on failure) or some triple whose first element is a proof, whose second element is an “update” environment

Δ , and show third argument is a set of constraints to be unified.

Note: The second argument, Δ , is redundant, as it is contained in the first argument (the proof). It is returned separately only to make pattern matching less cumbersome.

```
# gather_dec;;
- : env -> dec -> (proof * (string * expType) list * (expType * expType) list) option = <f
```

Note that in the previous setting, where you were trying to prove some $\Gamma \vdash e : \tau$, τ was an argument of the judgment. Restrictions were placed on τ using the constraints.

In this case, however, it is impossible to know Δ beforehand. You are not given Δ as an argument to `gather_dec`, and must instead generate and return a Δ .

The `val` rule is already implemented. Take a look at it for inspiration.

$$\frac{\Gamma \vdash e : \tau \mid C}{\Gamma \vdash \text{val } x = e \Rightarrow \{x \rightarrow \tau\} \mid C}$$

```
# let tree9 = LetIn(Valbind("x",Int 3), Id "x");;
val tree9 : exp = LetIn (Valbind ("x", Int 3), Id "x")
# print_proof (get_proof (infer tree9));;
```

```
{ } |= let (val x = 3) in (x) : int
|--{ } |= val x = 3 -> {x->int}
| |--{ } |= 3 : int
|--{x->int} |= x : int
```

```
- : unit = ()
```

7.1 Problems

7. (5 pts) Implement the `val rec` rule:

$$\frac{\{x \rightarrow \tau\} + \Gamma \vdash e : \tau \mid C}{\Gamma \vdash \text{val rec } x = e \Rightarrow \{x \rightarrow \tau\} \mid C}$$

```
# let tree16 = LetIn(ValbindRec("f",Fn("x",
      IfThenElse(App(Id "=",Pair(Id "x",Int 0)),
        String "finished",
        App(Id "f", App(Id "-",Pair(Id "x",Int 1))))
    )),
  App(Id "f",Int 3));;
```

```
...
```

```
# print_proof (get_proof (infer tree16));;
```

```
{ } |= let (val rec f = (fn x => if ((=)((x,0))) then ("finished") else ((f)((-)((x,1))))
|--{ } |= val rec f = (fn x => if ((=)((x,0))) then ("finished") else ((f)((-)((x,1))))
| |--{f->int -> string} |= (fn x => if ((=)((x,0))) then ("finished") else ((f)((-)((x
|   |--{x->int,f->int -> string} |= if ((=)((x,0))) then ("finished") else ((f)((-)((x
|   |--{x->int,f->int -> string} |= (=)((x,0)) : bool
|   | |--{x->int,f->int -> string} |= = : (int * int) -> bool
|   | |--{x->int,f->int -> string} |= (x,0) : (int * int)
```

```

|         |         |--{x->int,f->int -> string} |= x : int
|         |         |--{x->int,f->int -> string} |= 0 : int
|         |--{x->int,f->int -> string} |= "finished" : string
|         |--{x->int,f->int -> string} |= (f)((-)((x,1))) : string
|         |--{x->int,f->int -> string} |= f : int -> string
|         |--{x->int,f->int -> string} |= (-)((x,1)) : int
|         |--{x->int,f->int -> string} |= - : (int * int) -> int
|         |--{x->int,f->int -> string} |= (x,1) : (int * int)
|         |--{x->int,f->int -> string} |= x : int
|         |--{x->int,f->int -> string} |= 1 : int
|--{f->int -> string} |= (f)(3) : string
|--{f->int -> string} |= f : int -> string
|--{f->int -> string} |= 3 : int

```

- : unit = ()

8. (10 pts) Implement the local rule:

$$\frac{\Gamma \vdash d_1 \Rightarrow \Delta' \mid C_1 \quad \Delta' + \Gamma \vdash d_2 \Rightarrow \Delta \mid C_2}{\Gamma \vdash \text{local } d_1 \text{ in } d_2 \Rightarrow \Delta \mid C_1 \cup C_2}$$

```

# let tree17 = LetIn(Valbind("x",String "hi"),
                  LetIn(Local(Valbind("x",Int 5),Valbind("y",Id "x")),
                        Id "x"));;

```

...

```

# print_proof (get_proof (infer tree17));;

```

```

{} |= let (val x = "hi") in (let (local val x = 5 in val y = x) in (x)) : string
|--{} |= val x = "hi" -> {x->string}
| |--{} |= "hi" : string
|--{x->string} |= let (local val x = 5 in val y = x) in (x) : string
| |--{x->string} |= local val x = 5 in val y = x -> {y->int}
| | |--{x->string} |= val x = 5 -> {x->int}
| | | |--{x->string} |= 5 : int
| | |--{x->int,x->string} |= val y = x -> {y->int}
| | | |--{x->int,x->string} |= x : int
|--{y->int,x->string} |= x : string

```

- : unit = ()

9. (10 pts XC) Implement the sequence rule:

$$\frac{\Gamma \vdash d_1 \Rightarrow \Delta \mid C_1 \quad \Delta + \Gamma \vdash d_2 \Rightarrow \Delta' \mid C_2}{\Gamma \vdash d_1 d_2 \Rightarrow \Delta' + \Delta \mid C_1 \cup C_2}$$

```

# let tree15 = LetIn(Seq(Valbind("x",Int 3),Valbind("y",Int 4)),
                    App(Id "+",Pair(Id "x",Id "y")));;

```

...

```

# print_proof (get_proof (infer tree15));;

```

```

{} |= let (val x = 3 val y = 4) in ((+)((x,y))) : int
|--{} |= val x = 3 val y = 4 -> {y->int,x->int}
| |--{} |= val x = 3 -> {x->int}
| | |--{} |= 3 : int
| |--{x->int} |= val y = 4 -> {y->int}
| | |--{x->int} |= 4 : int
|--{y->int,x->int} |= (+)((x,y)) : int
|--{y->int,x->int} |= + : (int * int) -> int
|--{y->int,x->int} |= (x,y) : (int * int)
| |--{y->int,x->int} |= x : int
| |--{y->int,x->int} |= y : int

- : unit = ()

```