# M

The "m" algorithm first calls "m\_typedecls", which reads all the custom type declarations and places them in the environment.

Then it calls the recursive "m\_sccs", which starts checking the result of Tarjan's Strongly Connected Component algorithm.

# M\_typedecls

*See M.*

# M\_sccs

This function repeats the following actions for each SCC:

* It checks if the component contains interdependent variable declarations with “check\_scc”
* It creates a new environment for that specific SCC with “new\_env”
* It calls “m\_scc” to typecheck the current component in its new environment
* Adds the typed declarations to the old environment, adds arguments to function enties
* It adjusts the function (and function argument) types in the “old” environment according to the result of “m\_scc”

# Check\_scc

Since variables may not be dependent on eachothers, and a function cannot be dependent on a global variable that is dependent on that function, an SCC that contains a variable cannot contain anything else.

# New\_env

Generates a new environment by creating stubs for each function or variable in a Strongly Connected Component. Uses “pretype\_fun” and “pretype\_var” to take the predefined type into account if necessary.

# Pretype\_fun/Pretype\_var

*See New\_env.*

# M\_scc

For every declaration within the SCC that it is called with, “m\_scc” does the following:

* If it’s a function declaration, call “m\_fundecl”
* If it’s a variable declaration, call “m\_vardecl” with the type that is found in the environment

# M\_fundecl

* Checks for duplicate arguments
* Calls “m\_id\_fun” to check the environment for the current function
* Gives the arguments the correct types (with “type\_fargs”)
* Checks types for all local variables (with “m\_vardecl”)
* Calls “m\_stmts” with the typed locals added to the environment

# M\_vardecl

Just calls “m\_exp”, since the variable is already in the environment, its type merely needs to be unifiable to the one of its expression.

# M\_stmts

Recursively checks all statements in a list by calling “m\_stmt”. Interesting to note: the only way this can fail, is if one of the statements is something else than a Var. And the only way that can happen, is if it is or contains a return statement.

# M\_stmt

Needs “m\_fieldexp” for the “Stmt\_define” statement.

This function works pretty much as described in the college slides, except for the part that handles the match-statement. This particular statement works like this:

* It checks the type of the match-expression
* Then, for each case:
  + It places the hyperlocal variables from this case in a temporary environment
  + It tries to unify the type of the match-case with the match-expression (“m\_exp”)
  + It checks the “when” condition (“m\_exp” with as type Bool)
  + Finally, it checks the statementlist with “m\_stmts”

# M\_exp

Calls “m\_id\_fun” in the “Exp\_function\_call” expression.

Needs “m\_fieldexp” for the “Exp\_field” expression.

The difference with the college slides lies in the expressions that are needed for pattern-matching. These include the “Exp\_low\_bar” and “Exp\_constructor cons” expressions.  
“Exp\_low\_bar” is treated as a wildcard: with is fresh  
“Exp\_constructor” is handled by “m\_cons”:

# M\_fieldexp

The base-case for this recursive function is “Nofield id”, which represents the case in which there are no field operators (anymore). It is checked with “m\_id\_var”

When there are field operators left, it follows the function application example from the slide, treating the operator as a function and the rest of the expression as its argument. The operator is checked with “m\_field”.

# M\_field

Maps “Hd”, “Tl”, “Fst” and “Snd” to functions , , and respectively, with fresh.

# M\_cons, M\_id\_var, M\_id\_fun

Look in the environment for constructors, variables and functions respectively. Unify the given type with the type found in the environment. Return an error when the identifier isn’t found in the environment.