Martin Huyben (s…)

Tom Evers (s4205774)

Report

Compiler Construction

Table of Contents

[Table of Contents 1](#_Toc453947725)

[Lexing and parsing 2](#_Toc453947726)

[tokenizer.ml 2](#_Toc453947727)

[parser.ml 2](#_Toc453947728)

[exp\_parser.ml 3](#_Toc453947729)

[Typechecking 4](#_Toc453947730)

[m 4](#_Toc453947731)

[m\_main 4](#_Toc453947732)

[m\_typedecls 4](#_Toc453947733)

[m\_sccs 4](#_Toc453947734)

[check\_scc 4](#_Toc453947735)

[new\_env 5](#_Toc453947736)

[pretype\_fun/pretype\_var 5](#_Toc453947737)

[m\_scc 5](#_Toc453947738)

[m\_fundecl 5](#_Toc453947739)

[m\_vardecl 5](#_Toc453947740)

[m\_stmts 5](#_Toc453947741)

[m\_stmt 6](#_Toc453947742)

[m\_exp 6](#_Toc453947743)

[m\_fieldexp 6](#_Toc453947744)

[m\_field 6](#_Toc453947745)

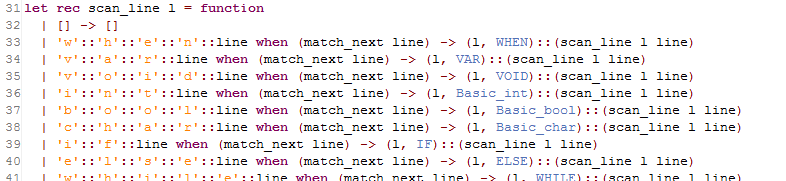
[m\_cons, m\_id\_var, m\_id\_fun 6](#_Toc453947746)

[Code generation 7](#_Toc453947747)

Lexing and parsing

# tokenizer.ml

This file contains functions for lexing a file. Files are scanned line by line. Every token is accompanied by the number of the line it was found on.



# parser.ml

Much more interesting would be the two parser modules, “parser.ml” and “exp\_parser.ml”. We decided to separate the expression parser from the rest of the parser, since our expression parser is kind of a beast. First, we will cover the normal part of the parser.

Our parser depends heavily on pattern matching, and in most cases works and behaves exactly as one would expect.

* First, it calls “remove\_comments” to clear all comments from the tokenlist
* Then, it calls “typedecllist\_parser” to generate a list of self-defined types
* Finally, it calls “decllist\_parser” to create the AST

The image below contains a so-called call-scheme. This scheme shows which functions are called by which. This should shed a light on how our parser works, better than a written report could do.

We could spend time trying to describe what each function does in detail, but looking at the code (combined with the knowledge in the call-scheme) gives a better understanding in our opinion.



# exp\_parser.ml

Typechecking

# graph\_make.ml

To enable multiple recursion, we had to do some preprocessing to group variable and function declarations in strongly connected components using Tarjan’s algorithm. This algorithm itself isn’t very interesting, as it is a literal translation from well-known code.

The part of the compiler that creates the nodes and edges is a bit more interesting, though doesn’t require much explanation either.



## Make\_graph

First, this function explicitly adds “read”, “write” and “isEmpty” to the graph.

Then, it calls “fv\_spl”.

## Fv\_spl

This function adds all the nodes to the graph. After that, it uses the other “fv”-functions to create edges from a declared function or variable, to all the function or variables it needs.

## The other fv-functions

These functions find all the identifiers used by the function or variable, that are not its own name.

The exceptions to this:

* Function arguments
* Local variables
* Hyperlocals in match-cases

The call-scheme for our typechecker:



# 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.

To conclude things, it checks just the main-function with “m\_main”.

# m\_main

Combination of “m\_sccs”, “m\_scc” and “m\_fundecl”. Checks a few things:

* ‘main’ is a function, not a variable
* ‘main’ has no arguments
* ‘main’ returns an integer

# 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 entries
* 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 each other, 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. This can be caused by the following things:

* If it concerns a global variable, “new\_env” did this, way back in “m\_sccs”
* If it concerns a local variable, “m\_fundecl” did this
* If it concerns a hyperlocal, “m\_stmt” did this

The variable’s type merely needs to be unifiable with 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.

Code generation

Other parts