MPRI course 2-4
Functional programming and type systems Programming task: Implementing Mini-Haskell

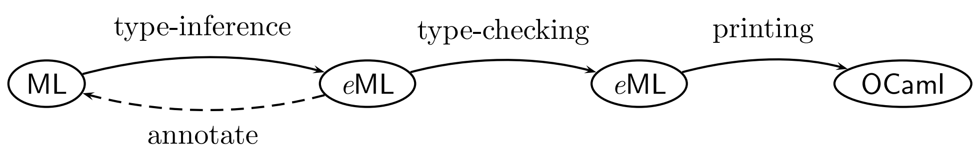
Yann Régis-Gianas and Didier Rémy

December 30, 2013

1 Summary

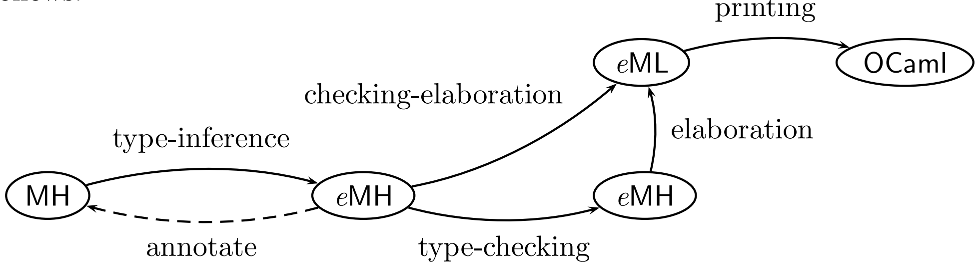
The purpose of this programming project is to implement the language Mini-Haskell described in the chapter on overloading—but restricting to ML style polymorphism, hereafter called MH.

We give you a working implementation of type inference for ML which moreover elaborate implicitly typed programs into explicitly typed programs, called eML. Programs of eML can be printed as OCaml programs to be compiled and run. They can also be printed back as some ML program—still retaining all explicit type information as type annotations (in fact eML is in bijection with a subset of ML where all function parameters, let-bindings and variables have explicit type annotations: they both carry the same amount of explicit type information but in different form.) This can be depicted as follows:



We provide implementations of both type inference on source terms and type-checking on explicitly typed terms of eML. This allows our compiler to taken either programs in ML (the default) or programs in eML. Explicitly typed programs are of course quite verbose, but the ability to write programs directly in eML may be quite useful, especially during the development, as well as a way of catching bugs: programs can be retyped-checked in eML after their elaboration from ML. Type-checking is implemented as the identity function, but it is only partial, since it fails when the source program is not well-typed—with appropriate error messages.

Your task is to extend this implementation to allow type classes in source terms, which we can depict as follows:



That is, type inference of ML must be extended to check well-typedness of new class constructs, returning an explicitly typed term in *e*MH, as well as typecheckng in *e*MH, so that elaboration can be performed on well-typed *e*MH programs translating them in to *e*ML. However, this two-step process can also be implemented as a single-step checking-and-elaboration algorithm. Finally, elaborated programs are *e*ML programs that can be printed as OCaml programs to be compiled and run.

Besides a running implementation of ML, a lexer, a parser and a printer for the full language are provided. Type inference is performed using constraint generation, much as our description of type inference in the course. Therefore, a constraint solver is already provided—but you will have to extend it to deal with dictionary constraints.

Section 3 gives an overview of the source files. Section 4 reminds the essential aspects of the design of *e*MH, the several restrictions made to simplify the design, and some implementation details or differences with the course notes. The detailed description can be found in the course notes. Section 5 describes the different tasks to be realized. The syntax of the full language is summarized in Appendix A.

2 Required software

The project can be implemented in any language of your choice, but we strongly recommend using OCaml, as the sources we provide are written in OCaml.

To use the sources we provide, you will need:

OCaml Any version $\geq$ 3.12 should do, but in doubt install version 4.01 from `http://caml.inria.fr/ocaml/rele` or from the packages available in your Linux distribution.

Linux, MacOS X, or Windows with the Cygwin environment The sources that we distribute were developed and tested under Linux. They should work under other Unix-like environments.

In addition, if you modify the parser (source file `parser.mly`), you will need the Menhir parser generator, available at `http://gallium.inria.fr/~fpottier/menhir/`. On linux and MacOS X, it can be installed via `opam`. The implementation uses a pretty printing library `pprint`, which is shipped jointly (in directory `pprint-20130324`) for convenience.

3 Overview of the provided sources

Sources can be found in the directory `src/`, which contains several four subdirectories `common`, `parsing`, `inference`, and `elaboration`.

common/* Several utilities.

common/errors.{ml, mli} A small utility for reporting errors

common/positions.{ml, mli} A small utility for dealing with positions in the source.

parsing/{tokens.{ml, mli},lexer.mll,parser.mly,prettyPrint} They define the concrete syntax for the language together with a pretty printer.

name.{ml, mli} Define types for the different sorts of names (identifiers, constructors, labels, and types)

types.{ml, mli} Defines the abstract syntax for types and type schemes and some utilities for manipulating them.

{AST,IAST,XAST}.{ml, mli} Defines the abstract syntax for ML and *e*ML. Both implementations are actually shared, using the common definitions in *AST*.

options.{ml, mli} This defines command line options

inference/ This is the largest part of the program: it implements the constraint solver, the type inference. It contains the following files:

inferenceTypes.{ml, mli}

typeAlgebra.{ml, mli}

typingEnvironment.{ml, mli}

typingExceptions.{ml, mli}

alphaRename.{ml, mli}

constraint.{ml, mli}

constraintGeneration.{ml, mli}

constraintSimplifier.{ml, mli}

constraintSolver.{ml, mli}

env.{ml, mli}

externalizeTypes.{ml, mli}

inferTypes.{ml, mli}

inferenceErrors.{ml, mli}

intRank.{ml, mli}

internalizeTypes.{ml, mli}

kindInferencer.{ml, mli}

multiEquation.{ml, mli}

unifier.{ml, mli}

elaboration This is the back-end of the program, responsible for the elaboration of type classes into dictionary passing. It contains the following files:

elaborateDictionaries.{ml, mli}

elaborationEnvironment.{ml, mli}

elaborationExceptions.{ml, mli}

elaborationErrors.{ml, mli}

front The top-level file of the program. Calls and combines the parser, the type-checker, the elaboration, and prints out the result as an OCaml program.

Makefile Build instructions. Issue the command "`make`" in order to generate the executable.

joujou The executable for the program (see Section §4.4).

In the `test/` directory, there are small programs written in MH, which you can give as arguments to `joujou` to see how they elaborate. Programs in the `test/good` subdirectory should pass the tests without errors. Programs in the `test/bad` subdirectory contain errors and should fail elaboration.

4 Language specification

The language MH is an extension of ML with type classes, quite similar to the language Mini-Haskell described in the course notes.

The main different is that we restrict source terms of MH to ML-like outermost polymorphism in MH, which implies that elaborated terms are themselves ML terms.

We refer to the course notes on overloading for the language specification and only remind here the key features of the language, the restrictions, and a few differences with the course notes.

4.1 The language ML

Our version of ML contains data types and record type declarations, pattern matching and recursive definitions. As in OCaml, a record type must contain at least one field and two records types cannot share the same label.

To help interact with an a OCaml we allow type and value external declarations. For example, a prelude may contain the following definitions:

$\texttt{type}\ \mathit{string} = \texttt{external}$
$\texttt{let}\ \mathit{create} : \mathit{int} \to \mathit{string} = \texttt{external}$ "String.create"
$\texttt{let}\ \mathit{get} : \mathit{string} \to \mathit{int} \to \mathit{char} = \texttt{external}$ "String.get"
$\texttt{let}\ \mathit{concat} : \mathit{string} \to \mathit{string} \to \mathit{string} = \texttt{external}$ "String.concat"

This populates the typing environment with a new (abstract) type `string` and three bindings (`create`, `get`, `concat`). The string following the internal will be used in elaborated code as the body of the type alias of let-binding. As a special case, the string may be omitted for external type declarations, which means that the type is primitive. In this case, the declaration will be dropped during elaboration.

4.2 The language MH

In the course, programs are sequences of class declarations $\vec{H}$ followed by instance declarations $\vec{h}$, and a final expressions M. First, we replace the final expression M by a sequence of ML toplevel declarations $\vec{d}$, which are either type declarations or let-bindings. We don't need a final expression as we can instead bind the final expression to a dummy variable.

We are more liberal and allow declarations of class declarations, instance definitions and toplevel definitions in any order. However, we do not allow reordering of definitions. We treat consecutive instance definitions recursively (see the paragraph below for their elaboration), two sequences of instance definitions separated by a toplevel or a class declaration are not recursive: the former is typed and elaborated without knowledge of the later. Except for consecutive instance declarations that are treated recursively, a declaration can only see declarations that were made earlier.

If one respects the strict ordering, we have the same semantics as in the course. However, we may also have a prelude of toplevel definitions or concatenate two independently developed libraries without having to reorder the code.

Elaboration of instance definitions

Although instance definitions are elaborated as recursive definitions, there are two restrictions to be made that have not been described in the course.

Assume that we are elaborating a sequence of instance definitions $\vec{h}$. The current typing environment is Γ_0 contains all previous definitions (classes, bindings, or previous sequences of instance definitions).

Each h will elaborate to a term N^h bound to a variable z_h of type σ_h. The expression N^h is of the form:

$$\Lambda\vec{\beta}\ \underbrace{\lambda(z_1 : K_1\ \alpha_1)\ldots\lambda(z_k : K_k\ \alpha_k)}_{\text{local context}}\ \Big\{\underbrace{u^K_{K'_1} = q_1, \ldots\ u^K_{K'_n} = q_n}_{\text{parent dictionaries}},\ \underbrace{u_1 = N^h_1, \ldots\ u_m = N^h_m}_{\text{methods}}\Big\}$$

The result of the elaboration of $\vec{h}$ is the evaluation context

$$\mathsf{let\ rec}\ \vec{z}_h : \vec{\sigma}_h = \vec{N}^h\ \mathsf{in}\ [\,]$$

exactly as defined in the course. This will extend the typing environment Γ_0 with new bindings $\Gamma_{\vec{h}}$ equal to $\vec{z}_h : \vec{\sigma}_h$. Since the definition is recursive, elaborated expressions will be typechecked in $\Gamma_0, \Gamma_{\vec{h}}$.

However, there are two restrictions to be made during elaboration: Let $\vec{h}$ be $\vec{h}', h, \vec{h}''$ and h be the instance definition currently under elaboration. Let $\Gamma_{\vec{z}}$ be the local context equal to $z_1 : K_1\ \alpha_1; \ldots z_p : K_p\ \alpha_p$. Then,

- Parent dictionaries are elaborated in the restricted context $\Gamma_0, \Gamma_{\vec{h}'}, \Gamma_{\vec{z}}$ so that they do not see the current instance definition h nor the remaining ones $\vec{h}''$.

  (This prevents parent dictionaries to be extracted from the current dictionary under construction, which would not make sense as it would be ill-founded.)

- Each method N^h_j is elaborated in the full context $\Gamma_{\vec{h}', h\vec{h}'}$ when N^h_j is a function, but in the restricted $\Gamma_0, \Gamma_{\vec{h}'}, \Gamma_{\vec{z}}$ otherwise.

  (This also prevents ill-founded recursion in the construction of the dictionary.)

After the elaboration of $\vec{h}$ the elaboration continues with the extended typing environment $\Gamma_o\Gamma_{\vec{h}}$, indeed. In particular, all previous instance definitions are always visible while elaborating an instance definition.

Identifiers By contrast with the course, we do not distinguish lexically ordinary variable from overloaded symbols (method names). Hence it is sometimes ambiguous whether an identifier is used as a variable or as a method name. To avoid confusion, we thus require that the same identifier is never used both as a variable and as a method name in the same program.

4.3 Restrictions

We restrict to single parameter type classes and non-overlapping instances, as in the course.

Types and type schemes

Types schemes in eMH are of the form $\forall\,(\vec{\alpha})\;\vec{P} \Rightarrow \tau$ where $\vec{P}$ is a canonical constraint, *i.e.* of the form $K_1\;\beta_1, \ldots K_p\;\beta_p$ where moreover, $\beta_i \neq \beta_j$ whenever K_i and K_j are related (equal or one is a superclass of the other).

Restriction on let-bindings

As in the course, we distinguish let-bindings of value-forms that can be generalized—and elaborated into overloaded identifiers and other let-bindings that are treated as monomorphic and cannot be overloaded. This prevents elaboration from changing the evaluation order (and is compatible with side having effects in the host language).

An in the course, we also immediately reject let-bound definitions whose type scheme has unreachable constraints, that is, constraints P in a type scheme $\forall\,(\vec{\alpha})\;[P]\tau$ that contains a variable α that does not appear free in its type.

We also request that types of toplevel let-bindings be closed. This implies that toplevel bindings of non-value forms should have ground types, using an explicit type annotation if necessary. This also prevent unresolved dictionary constraints at the toplevel.

Overloading on return type is allowed as long as the above restrictions are satisfied.

4.4 The different ways of using your program

The program should take source files in either MH or eMH, depending on the filename extension: "`./joujou` *filename.mlt*" expects an implicitly typed program while "`./joujou` *filename.mle*" expects an explicitly typed program. Both commands should output a program in "filename.ml" in OCaml syntax that can be compiled with version 4.01 of the compiler.

4.5 A sample program

See file `./test/elaboration/good/sample.mle` for the (an extended version) of the running example of the course.

5 Tasks

The programming project can be done as pair work, *i.e.* in teams of two people (*travail en binôme* in French). This implies a single submission and a single grade per team. However, we expect more work and more polished projects for pair-work submissions.

In particular, people working in team are required to do tasks 1, 2, 3, and 4, described below, while people working alone are only required to do tasks 1, 2, and 3. Every one may also do extensions for extra-credit. Task 4 will be considered as an extension for people working alone.

The 4 tasks are, in order of dependence:

Task 1 Extend the typechecker for eML to cover the full eMH language. This phase need not check that type-classes can be correctly elaborated, but this should verify the different restriction we made on type classes. The files to be modified are `elaborateDictionaries.ml` and `elaborationEnvironment.ml` in the `elaboration` directory. This can be tested on programs directly written in eMH.

Task 3 Implement elaboration of dictionaries. The file to be modified is `elaborateDictionaries.ml` in the `elaboration/` repository. This can be tested without having to implement the elaboration of terms. Indeed, a program that only declares instances without using them only need dictionary elaboration.

Task 3 Extend the typechecker for eMH to also perform the elaboration of type classes, and compile then away into a program expression in eML. The file to be modified is `elaborateDictionaries.ml`. The code can now be compiled by the OCaml compiler and run.

Task 4 Extend type inference in ML to perform type inference in MH and therefore transform programs from MH to eMH. The files to be modified are `constraintGeneration`, `constraintSolver`, and `constraintSimplifier` in the `inference` directory.

For extra credit You may explore any combination of the following improvements:

- Implement some medium size program that heavily relies on type classes.
- Relax some of the restrictions of the language. (*e.g.* allow for overlapping instances or associated types)
- Extend the program in any direction you are interested in.

6 Evaluation

Assignments will be evaluated by a combination of:

- Testing: your program will be run on the examples provided (in directory `test/`) and on additional examples.
- Reading your source code, for correctness and elegance.

7 What to turn in

When you are done, please e-mail `yrg@pps.univ-paris-diderot.fr` and `Didier.Remy@inria.fr` a `.tar.gz` archive containing:

- All your source files.
- Additional test files written in the small programming language, if you wrote any.

- If you implemented “extra credit” features, a `README` file (written in French or English) describing these additional features, how you implemented them, and where we should look in the source code to see how they are implemented.
- Please respect the initial layout of the archive. Decompressing the archive must produce a directory with your name and (at least) two subdirectories `src` and `test` equipped with `Makefile`s.

8 Deadline

Please turn in your assignment on or before **Saturday, March 1st 2014**.

A Concrete syntax of *e*MH

Whole programs:

$prog ::= (T \mid M \mid H \mid h^*)^*$

Class declarations:

$H ::=$ `class` $parents^?\ cvar\ tvar\ \{l_1 : \tau_1; \dots l_n : \tau_n\}$

Class identifiers:

$cvar ::= K$

Superclasses:

$parents$	$::= ctx$	with several superclasses

Instance definitions:

$h ::=$ `instance` $tps^?\ ctx^?\ cvar\ idx\ \{l_1 = M_1, \dots l_n = M_n\}$

Type parameters:

tps	$::= [tvar_1, \dots tvar_n]$	with type parameters

Typing context:

$ctx ::= cvar_1 tvar_1, \dots cvar_n\ tvar_n$ `=>`

Instance index:

idx	$::= tname$
	$\mid tvar\ tname$
	$\mid (tvar_1, \dots tvar_n)\ tname$

Expressions:

M	$::= x\ \bar{\tau}$	identifier
	$\mid M.l$	record projection
	$\mid C\ \bar{\tau}$	constant constructor
	$\mid C\ \bar{\tau}\ (M_1, \dots M_n)$	constructor with several arguments
	$\mid (M)$	parenthesized expression
	$\mid (M : \tau)$	type annotation
	$\mid [\alpha]M$	type abstraction
	$\mid$ `fun` $(x : \tau)$ `->` M	function with explicit type annotation
	$\mid M_1\ M_2$	application
	$\mid$ `match` M `with` p_1 `->` M_1 `|` `...` p_n `->` M_n	pattern-matching
	$\mid$ `let` $tps^?\ ctx^?\ (x : \tau) = M_1$ `in` M_2	non-recursive definitions
	$\mid$ `let rec` $tps^?_1\ ctx^?_1 f_1 : \tau_1 = M_1$	mutually recursive function definitions
	`...` `and` $tps^?_n\ ctx^?_n\ f_n : \tau_n = M_n$ `in` M	
	$\mid \{l_i = M_1, \dots, l_n = M_n\}\bar{\tau}$	records

Patterns:

p	$::= C\ \bar{\tau}$	constant constructor
	$\mid C\ \bar{\tau}\ (p_1, \dots, p_n)$	constructor with several arguments
	$\mid$ `_`	wildcard
	$\mid x$	variable
	$\mid p_1 \mid p_2$	disjunction

Type expressions:

τ	$::= tvar$	type variable
	$\mid \tau_1$ `->` τ_2	function type
	$\mid tname$	type constructor, no arguments
	$\mid \tau\ tname$	type constructor, one argument
	$\mid (\tau_1, \ldots \tau_n)\ tname$	type constructor
	$\mid (\tau)$	

Sequence of type applications:

$\bar{\tau}$	$::= [\tau_1,\ \ldots \tau_n]$	nonempty sequence of types

Type variables:

$tvar$	$::=$ `'`x	

Type definitions:

T	$::=$ `type` $tdef$ (`and` $tdef$)*	
$tdef$	$::= tname = tbody$	no parameters
	$\mid tvar\ tname = tbody$	one type parameter
	$\mid (tvar_1, \ldots, tvar_n)\ tname = tbody$	several type parameters

Type definitions:

$tbody$	$::= cstr$ `|` $\ldots cstr$	sum type
	$\mid \{\ell_1 : \tau_1; \ldots ; \ell_n : \tau_n\}$	record type

Constructor definitions:

$cstr$	$::= C$	constant constructor
	$\mid C$ `of` τ	with one argument
	$\mid C$ `of` τ_1 `*` $\ldots$ `*` τ_n	with several arguments