MPRI course 2-4
Functional programming and type systems
Programming project

Yann Régis-Gianas and Didier Rémy

December 20, 2012

1 Summary

The purpose of this programming project is to implement iML, an extension of an explicitly-typed version of ML with implicit arguments. We choose an explicitly typed version of ML for sake of simplification. The main task is the elaboration of source programs of iML into the ML subset of iML, which means resolving implicit arguments into ML subexpressions. The result of the elaboration should be a well-formed iML program that can be printed as an OCaml program, compiled, and run.

Section **??** describes the core subset of iML, its elaboration, and ambiguity resolution. The syntax of the full language is summarized in Appendix A.

The following parts of the program are provided: a lexer and parser, a constraint solver for first-order unification; a pretty-printer of iML programs as either iML or OCaml programs; and a front end that verifies well-formedness of the source program and performs alpha-conversion to prevent shadowing. We also provide an implementation of unification over types using a union-find data-structure, which may be used to check for non-overlapping of implicit definitions.

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.

2 Required software

To use the sources we provide, you will need:

OCaml Any version $\geq$ 3.10 should do, but in doubt install version 4.00.2 from `http://caml.inria.fr/ocaml/release.en.html` or from the packages available in your Linux distribution. (The source code should compile with version 3.10, but the code output by the prototype will not compile with earlier versions.)

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/`.

3 Overview of the provided sources

In the `src/` directory, you will find the following files:

ast.mli Defines the abstract syntax for the language.

printast.{ml, mli} Printing functions for types, expressions and programs.

settings.{ml, mli} Parses the command line and sets some flags.

error.{ml, mli} A small number of utilities for reporting errors.

type.{ml, mli} A small number of utility functions over the abstract syntax of types.

parser.mly, lexer.mll, error.{ml, mli} Parsing and error reporting. Together, the lexer and parser define the concrete syntax for the language.

stringMap.{ml, mli} Maps whose keys are strings. Useful for implementing various kinds of environments.

matching{ml, mli} Implements matching between types.

wf.{ml, mli} Checks well-formedness (well-scoping, arities, etc.) and performs alpha-conversion.

exproftype.{ml, mli} Implements the elaboration of types into expressions.

elaborate.{ml, mli} Implements the elaboration of iML into ML.

front.{ml, mli} The top-level file of the program. Calls and combines the parser, the well-formedness checker, the elaboration, and prints out the result of elaboration as an OCaml program.

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

joujou The executable for the program. Type "`./joujou` *filename.iml*" to type-check and elaborate the source program in *filename.iml*, and print the elaborated code as an OCaml source program into *filename.ml*. Add option "`-alpha`" to see the source code after alpha-conversion or "`-v`" to log some steps of the elaboration.

You will also find the implementation of unification over a union-find data structure that may be useful for checking for overlapping of implicit declarations or for some of the extensions.

unionFind.{ml, mli} Implements Tarjan's data structure for the union-find problem. This module underlies our implementation of first-order unification.

unification.{ml, mli} Implements first-order unification. This module defines the syntax of unification problems and uses them to check whether two types are unifiable.

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

4 Language specification

The language $i\mathsf{ML}$ is an extension of ML with implicit arguments. For simplification, we start with an explicitly-typed version of ML, so that types need only to be checked and not inferred. We only present the core of the language here. The full language has also datatypes, non-recursive definitions—see its syntax description in appendix **??**.

4.1 Types and expressions

We enriched arrow types with implicit arrows $\tau \Rightarrow \tau$ which describe functions of type $\tau \to \tau$, but whose argument will be left implicit at every use of f. We require that implicit arguments come first. Hence, we stratify the definition of types as follows:

$$\tau ::= \alpha \mid \tau \to \tau \qquad \rho ::= \tau \mid \tau \Rightarrow \rho \qquad \sigma ::= \rho \mid \forall\,(\alpha)\,\sigma$$

We also write $\bar{\tau} \Rightarrow \tau$ for $\tau_1 \Rightarrow \ldots \tau_n \Rightarrow \tau$. We often see this as an n-ary function and say that $\bar{\tau}$ is the domain and τ is the codomain of $\bar{\tau} \Rightarrow \tau$.

Core $i\mathsf{ML}$ extends core ML with three new constructs (ML expressions appear grayed on the first line).

$$\begin{aligned} M \;::=\; & x \mid \Lambda\alpha.M \mid M\,[\tau] \mid \mathsf{fun}\;(x:\tau) \to M \mid M\;M \mid \mathsf{let\;rec}\;x:\sigma = M\;\mathsf{in}\;M \\ & \mid \mathsf{let\;rec}\;?x:\sigma = M\;\mathsf{in}\;M \mid (?:\tau) \mid \mathsf{fun}\;(?x:\tau) \Rightarrow M \end{aligned}$$

We only consider recursive definitions in the core language. Types of definitions are provided, so that recursive definitions may be polymorphic. The full language has also non recursive definitions, which do not need a type annotation.

Actually, the primitive constructs for implicits are $\mathsf{let\;rec}\;?x : \sigma = M\;\mathsf{in}\;M$ and $(? : \tau)$. Implicit arguments are then added because they are quite handy but they can be translated away into explicit arguments with implicit values. The first construct is just a special form of the let-binding that declares the bound variable to be also available to build implicitly arguments. Typing environments carry this information along:

$$\Gamma \;::=\; \emptyset \mid \Gamma, \alpha \mid \Gamma, x:\tau \mid \Gamma, ?x:\tau$$

A binding $?x : \sigma$ is called an implicit binding. To factor out similar cases, we write εx for either x or $?x$. For instance, $\mathsf{let\;rec}\;\varepsilon x = M_1\;\mathsf{in}\;M_2$ stands for either $\mathsf{let\;rec}\;x = M_1\;\mathsf{in}\;M_2$ or $\mathsf{let\;rec}\;?x = M_1\;\mathsf{in}\;M_2$.

The construct $(? : \tau)$ is then used to build a value of the given type τ from implicit bindings. This mechanism is described more precisely below.

In functions, implicit parameters should always come before explicit parameters. Hence, one may write $\mathsf{fun}\;(?x_1 : \tau_1) \Rightarrow \mathsf{fun}\;(x_2 : \tau_2) \to M$ but not $\mathsf{fun}\;(x_2 : \tau_2) \to \mathsf{fun}\;(?x_1 : \tau_1) \Rightarrow M$. This is not enforced in the syntax of expression, but in the typing rules (and the syntax of types). That is, types of implicit functions $\bar{\tau} \Rightarrow \tau$ are not first-class.

4.2 Elaboration of expressions

Programs with implicit arguments can be elaborated into programs of core ML by synthesizing the missing arguments from their types. The elaboration is described on Figure **??**. The elaboration is

$$\textsc{Var}\ \frac{\varepsilon x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma \rightsquigarrow x} \qquad \textsc{Tabs}\ \frac{\Gamma, \alpha \vdash M : \rho \rightsquigarrow N}{\Gamma \vdash \Lambda\alpha.M : \forall\,(\alpha)\ \rho \rightsquigarrow \Lambda\alpha.N} \qquad \textsc{Tapp}\ \frac{\Gamma \vdash M : \forall\,(\alpha)\ \rho \rightsquigarrow N}{\Gamma \vdash M\,[\tau] : [\alpha \mapsto \tau]\rho \rightsquigarrow N\,[\tau]}$$

$$\textsc{App}\ \frac{\Gamma \vdash M_1 : \tau_2 \to \tau_1 \rightsquigarrow N_1 \qquad \Gamma \vdash M_2 : \tau_2 \rightsquigarrow N_2}{\Gamma \vdash M_1\ M_2 : \tau_1 \rightsquigarrow N_1\ N_2}$$

$$\textsc{Abs}\ \frac{\Gamma, x : \tau_2 \vdash M_1 : \tau_1 \rightsquigarrow N_1}{\Gamma \vdash \mathsf{fun}\ (x : \tau_2) \to M_1 : \tau_2 \to \tau_1 \rightsquigarrow \mathsf{fun}\ (x : \tau_2) \to N_1}$$

$$\textsc{LetRec}\ \frac{\Gamma, \varepsilon x : \sigma_1 \vdash M_1 : \sigma_1 \rightsquigarrow N_1 \qquad \Gamma, \varepsilon x : \sigma_1 \vdash M_2 : \sigma_2 \rightsquigarrow N_2}{\Gamma \vdash \mathsf{let\ rec}\ \varepsilon x : \sigma_1 = M_1\ \mathsf{in}\ M_2 : \sigma_2 \rightsquigarrow \mathsf{let\ rec}\ x = N_1\ \mathsf{in}\ N_2} \qquad \textsc{Implicit}\ \frac{\Gamma \vdash \tau \rightsquigarrow N}{\Gamma \vdash (? : \tau) : \tau \rightsquigarrow N}$$

$$\textsc{Abs-Implicit}\ \frac{\Gamma, ?x : \tau \vdash M : \rho \rightsquigarrow N}{\Gamma \vdash \mathsf{fun}\ (?x : \tau) \Rightarrow M : \tau \Rightarrow \rho \rightsquigarrow \mathsf{fun}\ (x : \tau) \to N} \qquad \textsc{App-Implicit}\ \frac{\Gamma \vdash M : \tau \Rightarrow \rho \rightsquigarrow N \qquad \Gamma \vdash \tau \rightsquigarrow N_0}{\Gamma \vdash M\ : \rho \rightsquigarrow N\ N_0}$$

Figure 1: Elaboration of programs

defined by translation of typing derivations and thus simultaneously describes the typing of iML. It is almost straightforward, because the elaboration of terms uses an auxiliary judgment $\Gamma \vdash \tau \rightsquigarrow N$ for the elaboration of types into expressions, which is the hardest part of elaboration, described below.

Functions with implicit arguments are described by rule Abs-Implicit. The counter-part is rule App-Implicit which passes arguments implicitly when needed. However, to prevent the application of rule App-Implicit immediately after rule Abs-Implicit which would resolve the implicit argument in the definition context instead of the invocation context, we restrict the use of rule App-Implicit to terms M that are either variables or type applications. Moreover, we request that rule App-Implicit be used whenever it applies, *i.e.* implicit arguments should also be resolved as soon as they can. Hence, when a variable or a type application has type $\tau_1 \Rightarrow \ldots \tau_k \Rightarrow \tau$, then its k implicit arguments are elaborated immediately.

Notice that a function with an implicit argument $\mathsf{fun}\ (?x : \tau) \Rightarrow M$ also treats its argument as an implicit definition in M. In particular, we cannot write $\mathsf{fun}\ (x : \tau) \Rightarrow M$ in the syntax which would mark x as an implicit argument that is not an implicit definition. While treating the parameter as an implicit definition is often useful, allowing this systematically rarely entails a problem (such as creating undesired ambiguities).

Notice that while implicit bindings are visible in the typing context and can be used explicitly, their implicit arguments cannot be passed explicitly and must always be inferred.

Type well-formedness Well-formedness for types and contexts is standard.

$$\textsc{Elab-Var}\ \frac{?x : \sigma \in \Gamma}{\Gamma \vdash \sigma \rightsquigarrow x} \qquad \textsc{Elab-Tapp}\ \frac{\Gamma \vdash \forall\,(\alpha)\,\rho \rightsquigarrow N}{\Gamma \vdash [\alpha \mapsto \tau]\rho \rightsquigarrow N\,[\tau]} \qquad \textsc{Elab-App}\ \frac{\Gamma \vdash \tau \Rightarrow \rho \rightsquigarrow N_1 \qquad \Gamma \vdash \tau \rightsquigarrow N_2}{\Gamma \vdash \rho \rightsquigarrow N_1\,N_2}$$

Figure 2: Elaboration of types

4.3 Elaboration of types into expressions

The elaboration of a type τ is performed in a typing context Γ and returns an expression N of type τ as described by the rules of Figure **??**. (Notice that elaboration uses only bindings declared as implicit in Γ.) In fact, these three rules may be factored into a single rule:

$$\textsc{Elab}\ \frac{?x : \forall\,(\bar\alpha)\,\bar\tau \Rightarrow \tau \in \Gamma \qquad [\bar\alpha \mapsto \bar\tau']\tau = \tau_0 \qquad \Gamma \vdash [\bar\alpha \mapsto \bar\tau']\tau_i \rightsquigarrow N_i}{\Gamma \vdash \tau_0 \rightsquigarrow x\,[\bar\tau']\,\bar N}$$

Hence, one may see implicit definitions as deduction rules, and the elaboration as applying rules in a prolog like fashion. Elaborated terms lie in the subset of expressions of the form:

$$N ::= x\,[\bar\tau]\,\bar N$$

Hence, every node may be labeled by an implicit definitions and the type at which it is used. A path in an elaborated term is then a sequence of $x_1^{\tau_1} \dots x_n^{\tau_n}$.

Given a context Γ and a type τ, the elaboration of τ may be undefined in case there is no M such that $\Gamma \vdash \tau \rightsquigarrow M$. We may distinguish two sources of failure: an attempt to build a derivation may fail because in any partial derivation there is at least one leaf where no rule applies; on the contrary, it may happen that any partial derivations can be extended for ever. Below we restrict the definition of well-formed elaborated terms, so that we can easily rule out this second situation.

When the elaboration succeeds, there may be a unique M or multiple solutions. When there are multiple solutions, we say that the elaboration is ambiguous.

Restriction on the types of implicit definitions Rule Elab may be read algorithmically as long as variables $\bar\alpha$ all appear free in τ so that the matching $[\bar\alpha \mapsto \bar\tau']\tau = \tau_0$ completely determines the substitution $[\bar\alpha \mapsto \bar\tau']$ from τ_0. Then the elaboration recursively elaborate the arguments and backtrack in case of failure. Otherwise, one would need to guess (part of) the substitution $[\bar\alpha \mapsto \bar\tau']$ so that the elaboration of the arguments may succeed.

By default, we will restrict to "algorithmic" rules and will *reject implicit declarations that do not satisfy this condition.* The general case may later be considered as an extension, activated by an option on the command line.

4.4 Termination

To ensure termination of elaboration, we reject elaborated terms that would contain paths of the form $\dots x_i^{\tau_i} \dots x_j^{\tau_j} \dots$ such that either

- $\tau_i = \tau_j$, or
- $x_j = x_i$ and $|\tau_j| > |\tau_i|$ where $|t|$ is the size of τ counted as the number of type constructors in τ (including arrows).

Notice that the second criterion allows types to temporarily grow as long as we use different implicits and also allows reusing the same rules as long as types strictly decrease.

Terms that do not satisfy this criterion should not be considered as solution of elaboration. Therefore, *you must implement the combination of both criteria.*

Notice that a looser termination criterion (that would allow more elaborations, but still terminates) will be considered incorrect as this will change the resolution of ambiguities, hence the result of elaboration: indeed, even a term M that verifies $\Gamma \vdash \tau \rightsquigarrow M$ but does not satisfy the termination criterion is not considered as a solution and is simply discarded when resolving ambiguities.

4.5 Ambiguities

Ambiguous solutions are a source of non determinism. The programmer must be able to choose if they should be rejected or if they should be resolved with a clear deterministic policy. In this section, we motivate and formally define three different policies.

Non-overlapping of implicit definitions Ambiguities may be avoided if implicit definitions are never overlapping. Two definitions $x : \sigma$ and $x' : \sigma'$ (with $x \neq x'$) are overlapping if the codomains of σ_1 and σ_2 are unifiable (with respect to variables that are bound in σ_1 and σ_2).

Allowing ambiguities Requiring that implicit declarations are never ambiguous is too restrictive in practice, so we must tolerate some ambiguities and resolve them. Although implicit definitions may be used to perform arbitrary computation, including side effects, we may assume a disciplined use of implicits—under the user's responsibility.

A typical use of implicits would be to simulate a type class mechanism *via* implicit dictionary construction. In this setting, implicit definitions are of two kinds: constructors that build dictionaries of a given type and accessors that extract subdictionary of some type from bigger dictionaries. While it is good usage that constructions do not overlap, so that dictionaries of a given type are unique, accessors may remain ambiguous: they will allow several ways of retrieving the same dictionary from different places. (This is actually not fully accurate because of local scoping of constructors, which allows to non ambiguously construct non coherent dictionaries in two different scopes.)

Resolving ambiguities Ambiguities may be resolved using a total ordering of implicit bindings. The default ordering (which we will refine later) is to consider implicit bindings in the order of their introduction in the typing context, giving higher priority to the those introduced last.

From a total ordering on implicit bindings, we define the following partial ordering on elaborated terms:

$$\frac{x \prec x'}{x[\bar{\tau}]\bar{M} \prec x'[\bar{\tau}']\bar{M}'} \qquad \frac{M_1 \prec M_1' \quad \dots \quad M_n \prec M_n'}{x[\bar{\tau}]\ M_1 \dots M_n \prec x[\bar{\tau}]\ M_1' \dots M_n'}$$

Two elaborated terms of the same type always have a minimal element for this partial ordering: choose the toplevel rule of higher priority, which determines the type parameters $\bar{\tau}$, hence the type τ_i at which each subterm M_i should be elaborated, which inductively have a minimal solution.

Lexicographic ordering Still, allowing overlapping bindings and resolving them by a total ordering is not satisfying: as explained above, constructors should not overlap.

Thus we propose to classify bindings in three groups, using three explicit *policies* indicated in the syntax as an annotation to the question mark:

- Higher priority bindings (?>): these may overlap with any other binding and are resolved by the introduction ordering.
- Normal bindings (?): these should not overlap with any other normal binding. There are used if no higher priority binding applies.
- Lower priority bindings (?<): these may overlap with any other binding and are resolved by the inverse introduction ordering: those introduced last have lower priority.

Hence, the ordering of bindings is the lexicographic ordering of the groups $(low, normal, high)$.

4.6 Beyond the core language

We have just presented the core language, formally. The language we implement has datatypes (sums of products), non recursive let bindings. While we have allowed recursive definitions, we restrict them to functions in a call-by-value language. Hence, in the implementation we also have non-recursive (explicit and implicit) definitions of arbitrary expressions.

5 Tasks

The 4 tasks are in order of dependence.

Task 1 Implement a typechecker for the subset of the source language that does not contain implicits. The file to be modified is mainly `elaborate.ml`.

Task 2 Implement the elaboration of types. (The files to be modified are `exproftype.ml` and `matching.ml`.) Complete the elaboration of implicit expressions. (The file to be modified is `elaborate.ml`.) You may ignore detection of non-overlapping bindings at first—and resolve normal bindings in order of introduction.

Task 3 Implement detection of overlapping for normal bindings.

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

- Write an efficient implementation of the structure `Exproftype.Dn`, storing implicit definitions according to the structure of their codomain so that finding a candidate for a type can be computed by exploring the structure in time proportional to the sum of the sizes of the matching solutions.
- Implement type inference so that most type annotations in the source program can be dropped. (Types of implicit bindings should remain explicit.)
- Write programs in *i*ML that make interesting uses of implicit arguments.
- 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 `Didier.Remy@inria.fr` and `yrg@pps.univ-paris-diderot.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) to subdirectories `src` and `test` equipped with `Makefile`s.

8 Deadline

Please turn in your assignment on or before **Friday, March 1st 2013**.

A Concrete syntax

Whole programs:

prog	::= $tdef^*$ `program` M	type declarations + main expression

Expressions:

M	::= x	identifier
	| $C\ \bar{\tau}$	constant constructor
	| $C\ \bar{\tau}\ M$	constructor with one argument
	| $C\ \bar{\tau}\ (M_1, \ldots M_n)$	constructor with several arguments
	| (M)	parenthesized expression
	| `fun` $(x : \tau)$ `->` M	function
	| $M_1\ M_2$	application
	| `begin match` M `with` p_1 `->` M_1 `|` $\ldots$ p_n `->` M_n `end`	pattern-matching
	| `let` $\varepsilon x = M_1$ `in` M_2	non-recursive definitions
	| `let rec` $\varepsilon_1 f_1 : \sigma_1 = M_1$ $\ldots$ `and` $\varepsilon_n f_n : \sigma_n = M_n$ `in` M	mutually recursive function definitions
	| $(? : \tau)$	implicit expression
	| `fun` $(?\diamond x : \tau)$ `=>` M	function with implicit argument
	| `Lam` $tvar_1 \ldots tvar_n \bullet M$	type abstractions
	| $M[\tau_1, \ldots \tau_n]$	type applications

Patterns:

p	::= $C\ \bar{\tau}$	constant constructor
	| $C\ \bar{\tau}\ x$	constructor with one argument
	| $C\ \bar{\tau}\ (x_1, \ldots, x_n)$	constructor with several arguments

Type expressions:

τ	::= *tvar*	type variable
	| τ_1 `->` τ_2	function type
	| *tname*	type constructor, no arguments
	| τ *tname*	type constructor, one argument
	| $(\tau_1, \ldots \tau_n)$ *tname*	type constructor, several arguments
	| (τ)	

Sequence of type applications:

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

Binding:

ε	::=	explicit definition
	| $?\diamond$	implicit definition

Priority:

$\diamond$	::=	normal
	| `>`	higher
	| `<`	lower

Type variables:

tvar ::= `'`*x*

Type definitions:

tdef ::=	`type` *tname* = *cstr* `|` ... *cstr*	no parameters
|	`type` *tvar tname* = *cstr* `|` ... *cstr*	one type parameter
|	`type` $(tvar_1, \ldots, tvar_n)$ *tname* = *cstr* `|` ... *cstr*	several type parameters

Constructor definitions:

cstr ::=	C	constant constructor
|	C `of` τ	with one argument
|	C `of` τ_1 `*` ... `*` τ_n	with several arguments