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

François Pottier

2016–2017

An up-to-date version of this document can be found at:

http://yann.regis-gianas.org/mpri/projet-mpri-2-4-2.pdf.

1 Summary

The purpose of this programming project is to implement:

1. a type-checker for System F with generalized algebraic data types (GADTs);
2. type-preserving defunctionalization, expressed as a transformation of System F with GADTs into itself.

The type-checker will be used to check the program before and after the translation.

2 Task description

The project is closely based on the paper "Polymorphic Typed Defunctionalization and Concretization", by Pottier and Gauthier [1]. Sections 1, 2, 3, and 6 of the paper are required reading. Please study them in detail.

Task 1 Implement the type-checker. The file to modify is `typecheck.ml`.

System F with GADTs is described in Section 2 of the paper. Look at the syntaxes of types, type schemes (which describe data constructors), and terms. Compare them with the definitions that are supplied in `src/types.mli` and `src/terms.ml`. Up to a few minor differences, they are identical. Make sure that you understand these definitions. If something is unclear, do not hesitate to ask questions; the earlier you ask, the better.

Some of the differences between the paper and the code include:

- the paper has a let rec construct, while the code has separate let and fix constructs;
- the paper distinguishes types and type schemes; in the code, the distinction is conceptually present, but, in order to avoid code duplication, the two syntactic classes are merged, so an `ftype` can represent either a type or a type scheme; a couple of constructs, namely `TyWhere` and `TyTuple`, are permitted only within type schemes;
- the paper uses record labels to identify the fields of a data constructor, while the code gets rid of record labels and views the fields as an ordered sequence—a tuple;
- the concrete syntax is sometimes different: for instance, the concrete syntax of the case analysis construct is "match t return T with $\bar{c}$ end", where t is a term and T is a type; see `parser.mly` and the sample code in `test/good/*.f` for details.

Task 2 Implement defunctionalization. The file to modify is `defunct.ml`.
Defunctionalization is described in section 3 of the paper.

For extra credit If you wish to go further, you can improve the type-checker and/or improve defunctionalization. Here are a few suggestions:

- (moderately difficult) as suggested in Section 6 of the paper, generate code for specialized versions of *apply*, for use at call sites where multiple arguments are supplied and/or arguments of a known type are provided;
- (moderately difficult) think about and optimize the treatment of toplevel function definitions; if the function f is defined at toplevel, then f should not be defunctionalized, and a reference to f should never be captured in a closure;

Other improvements are welcome.

3 Required software

To use the sources that we provide, you will need:

Objective Caml Any reasonably recent version of OCaml should do. If in doubt, install version 4.04 from `http://caml.inria.fr`, from the packages available in your Linux distribution or from OPAM.

The Menhir parser generator Available at `http://gallium.inria.fr/~fpottier/menhir/` or from OPAM. This tool is required in order to produce `parser.mli` and `parser.ml` out of `parser.mly`.

Linux, FreeBSD, MacOSX, or some other Unix-like system Notice that the `Makefile`s that we distribute have not been tested under Microsoft Windows.

4 Overview of the provided sources

Many components of the system are provided, including: definitions of the syntaxes of types and terms; a lexer and parser; a pretty-printer for types and terms; code for dealing with binders; code for dealing with types, including equality and entailment checks; code for reporting type errors.

This code is briefly described in the list below. The modules at the beginning of the list, up to `pprint`, are generic, and could be used without changes in type-checkers for other languages. The rest of the modules, beginning with `syntax`, are specific to our little language.

The files that you should study first are `types.mli` and `terms.mli`. These files define the internal representations of types and terms. You will work with these representations.

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

pllib/identifierChop.mll, pllib/identifier.{ml, mli} Identifiers. An identifier is essentially a pair of a sort and a string. Each sort defines a disjoint namespace.

pllib/atom.{ml, mli} Atoms. An atom is the internal object used to represent a name.

pllib/error.{ml, mli} Generic error reporting facilities.

pllib/import.{ml, mli} Generic facilities for converting identifiers to atoms and detecting unbound identifiers.

pllib/export.{ml, mli} Generic facilities for converting atoms back to identifiers, while avoiding unintentional capture.

pllib/lexerUtil.{ml, mli} Generic utilities for lexical analysis.

pprint.{ml, mli} Generic pretty-printing facilities.

syntax.ml Abstract syntax for types and terms. This syntax is produced by the parser. In this version of the syntax, names are represented as identifiers.

parser.mly, lexer.mll Together, the lexer and parser define the concrete syntax for the language.

types.{ml, mli} Internal representation of types, up to α-conversion. This representation is used by the type-checker.

terms.ml Internal representation of terms. This representation is used directly by the type-checker. It also serves as both the source and target languages of the translation.

symbols.{ml, mli} Offers a few auxiliary functions that help work with terms.

internalize.{ml, mli} Checks that all identifiers are bound, and replaces identifiers with unique atoms, so as to produce an internal representation of types and terms. This code comes after the parser.

print.{ml, mli} Pretty-printers for types and terms.

typerr.ml Auxiliary functions that help report type errors during typechecking.

unionFind.{ml, mli} This data structure is used by the `equations.ml`. You don't need it directly.

equations.{ml, mli} A decision procedure for entailment between type equations. This algorithm answers questions of the form: does C_1 entail C_2? where C_1 and C_2 are constraints. This is written $C_1 \Vdash C_2$ in the paper.

typecheck.{ml, mli} The type-checker. **This file is incomplete.**

defunct.{ml, mli} An implementation of defunctionalization. **This file is incomplete.**

main.ml This driver interprets the command line and invokes the above modules as required.

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

joujou The executable file for the program. Type "`./joujou` *filename*" to process the program stored in *filename*. Use the options "`-echo`" to echo the source program; "`-typecheck`" to typecheck the source program and print its type; and "`-translate`" to typecheck the source program, translate it, and print the translated program.

In the `test/` directory are small programs written in our functional language, which you can give as arguments to `joujou`. The programs in the `test/good` subdirectory are well-typed and should be silently accepted by the type-checker. The programs in the `test/bad` subdirectory contain type errors and should be rejected by the type-checker. The programs in the `test/warnings` subdirectory contain case analyses with missing clauses or redundant clauses and should be accepted with warnings.

The files `test/warnings/*.err` and `test/bad/*.err` are produced by our reference implementation of the type-checker. These files are provided to help you understand why the sample programs are incorrect. The warnings and error messages produced by your implementation need not be identical to these sample messages.

In order to test your implementation, run "`make test`". This invocation of the `Makefile` submits all the files of the form `test/good,warnings,bad/*.f` to your type-checker, and checks that the outcomes are appropriate. The script then submits the files `test/good/*.f` to your translator. It checks that the translation succeeds and that the output is again a well-typed program. Hint: test your type-checker thoroughly before trying to implement defunctionalization.

Advice Quite obviously, we require that you complete task 1 (the type-checker) before proceeding to task 2 (defunctionalization).

We strongly recommend that you regularly take checkpoints (that is, snapshots of your work) so that you can later easily roll back to a previous consistent state in case you run into an unforeseen problem. Using a versioning tool such as `git` or `svn` is recommended.

5 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. Make sure that "`make test`" succeeds!
- Reading your source code, for correctness and elegance.

6 What to turn in

When you are done, please e-mail to Yann Régis-Gianas 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.

7 Deadline

Please turn in your assignment on or before **Friday, 24 February 2017**.

References

[1] François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, 19:125–162, March 2006.