MPRI course 2-4-2
"Functional programming languages"
Programming project

Xavier Leroy and François Pottier

December 11, 2006

1 Summary

The purpose of this programming project is to implement an interpreter, a typechecker, and a compiler (down to a simple abstract machine) for a small functional programming language equipped with (possibly parameterized) algebraic data types. The following parts of the program are provided: a lexer and parser, a constraint solver for first-order unification constraints, an abstract machine, and its execution engine.

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

2 Required software

To use the sources we provide, you will need:

Objective Caml Any version $\geq$ 3.0 should do, but in doubt install version 3.09.3 from `http://caml.inria.fr/ocaml/release.en.html` or from the packages available in your Linux distribution.

The Menhir parser generator Available at `http://gallium.inria.fr/~fpottier/menhir/`. This tool is required in order to produce `parser.mli` and `parser.ml` out of `parser.mly`. (For those who don't want to install Menhir, we do provide `parser.mli` and `parser.ml`, but you will need to modify the `Makefile` in order to let `make` know that these files are not generated and should not be destroyed.)

Linux, FreeBSD, MacOSX, or some other Unix-like system The `Makefile` that we distribute has not been tested under Microsoft Windows. You are on your own if you insist on using Windows.

3 Overview of the provided sources

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

abstractSyntax.mli Defines the abstract syntax for the language.

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.

option.{ml, mli} Various utility functions for values of type `'a option`.

print.{ml, mli} Various utility functions for pretty-printing.

wf.{ml, mli} Check that a program is well-formed (no unbound variables, etc.).

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, which the constraint generator must produce.

generator.{ml, mli} Specifies the constraint generator. The implementation is missing; to be completed in task 3.

interpreter.{ml, mli} The skeleton of the interpreter. To be completed in task 1.

machine.{ml, mli} Definition of the abstract machine: instruction set and execution engine for abstract machine code.

compiler.{ml, mli} The skeleton of the compiler. To be completed in task 2.

settings.{ml, mli} Parses the command line.

front.{ml, mli} The top-level file of the program. Calls and combines the parser, the type-checker, the interpreter, the compiler and the execution engine of the abstract machine.

Makefile, Makefile.auto, Makefile.shared, ocamldep.wrapper Build instructions. Issue the command "`make`" in order to generate the executable.

joujou The executable for the program. Type "`./joujou` *filename*" to type-check, execute and compile the program in *filename*.

In the `test/` directory are small programs written in our functional language, which you can give as arguments to `joujou` to see how they execute.

4 Tasks

Task 1 Implement an interpreter for the source language. The file to modify is `interpreter.ml`.

Task 2 Implement a compiler from the source language to the abstract machine. The file to modify is `compiler.ml`.

Task 3 Study the specification that the constraint generator must meet, which is found in `generator.mli`, and implement the generator. The file to modify is `generator.ml`.

At the moment, the generator is incomplete and always produces an empty unification problem, which means that the inferred type is always "$\forall\alpha.\alpha$". It is up to you to construct a unification problem that is necessary and sufficient for the code to be well-typed.

Note that, for simplicity, we only implement *simple* (that is, *monomorphic*) type inference: no generalization will be performed at `let` constructs. Only the data constructors can have (closed) polymorphic type schemes, which are given by the `dcenv` parameter.

In order to better understand the entire type inference process, it is recommended, although not strictly necessary, to have a look at the modules **UnionFind** and **Unification**, which perform constraint solving.

For extra credit Extend the program in any direction you're interested in: additional language features, polymorphic type inference, understandable type error messages, compiler optimizations, ...

5 What to turn in

When you are done, please e-mail `Xavier.Leroy@inria.fr` and `Francois Pottier@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.

6 Deadline

Please turn in your assignment on or before **Sunday, 25 February 2007**.