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

Xavier Leroy and François Pottier

December 19, 2007

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 exceptions and (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 3.10 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`.

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.

exnconv.{ml, mli} The skeleton of the monadic conversion of exceptions. To be completed in task 4.

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, interpret, compile and execute the compiled code for the program in *filename*. Add option "`-t`" to obtain a detailed trace of the execution of the compiled code.

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. Programs in the `test/good` subdirectory should pass type-checking and execute without errors. The expected result value is given at the end of each source file. Programs in the `test/bad` subdirectory contain type errors and should fail type-checking.

4 Tasks

The 4 tasks are independent and can be completed in any order. We recommend to start with task 1 to familarize yourself with the source language.

Task 1 Implement an interpreter for the source language. The file to modify is `interpreter.ml`. The interpreter must follow a call-by-value, left-to-right evaluation strategy.

We recommend to use environments and function closures as in the "canonical efficient interpreter" of Leroy's lecture 1. To interpret exceptions (the `raise` and `try...with` constructs), a simple approach is to use Caml's exception mechanism: interpreting `raise` e raises the Caml exception `UncaughtException`(v) where v is the value of expression e; likewise, the interpretation of `try` e_1 `with` $x \to e_2$ uses Caml's `try...with` construct to catch `UncaughtException` exceptions raised during the interpretation of e_1. An equally valid alternative is to write the interpreter in purely functional style, so that it returns `V`(v) if an expression terminates normally with value v, and `E`(v) if an expression terminates prematurely on an uncaught exception v.

Task 2 Implement a compiler from the source language (excluding the `raise` and `try...with` constructs) to the abstract machine. The file to modify is `compiler.ml`.

First, study the instruction set of the abstract machine (file `machine.mli`). It extends the Modern SECD of Leroy's lecture 2 with a few instructions to deal with data constructors and pattern-matching. However, it provides no instructions to raise and catch exceptions, which is why the compiler should just fail when it encounters a `raise` or `try...with` in the source program.

Notice that the abstract machine uses de Bruijn indices (positions) to access its environment, while the source language uses names for variables. One possibility is to write a pre-compilation pass that translates the source language to a similar language where variables are identified by de Bruijn indices. It is simpler, however, to do this translation on the fly during compilation. To this end, the compilation functions should take as extra argument a compilation environment containing the names of all source variables currently in scope and from which their de Bruijn indices can easily be computed.

The compilation of an expression should follow the general pattern of Leroy's lecture 2. The code generated for an expression should leave its value on top of the stack, preserving the environment and the values initially present on the stack. For the `match` a `with` ... construct, a cascade of `IIfconstr` over the scrutinee a shoud be generated, one for each case of the pattern-matching. When a case $C(x_0, \ldots, x_n) \to b$ matches, a sequence of "IField" and "ILet" instructions is needed to extract the values $v_0, \ldots, v_n$ of the arguments of constructor C and bind them to variables $x_0, \ldots, x_n$.

Task 3 Complete the implementation of type inference for the source language. 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 the `raise` and `try...with` constructs, the typing rules that the type inference engine should implement are the following:

$$\frac{\Gamma \vdash a : \tau_{exn}}{\Gamma \vdash \texttt{raise}\ a : \tau} \qquad \frac{\Gamma \vdash a_1 : \tau \quad \Gamma,\ x : \tau_{exn} \vdash a_2 : \tau}{\Gamma \vdash (\texttt{try}\ a_1\ \texttt{with}\ x \to a_2) : \tau}$$

The type τ_{exn} of exception values can be any fixed type: that is, a single type τ_{exn} must be used in all `raise` and `try...with` constructs present in the program. The identity of the type τ_{exn}, however, is not important, and can be inferred: just use a type variable, shared between all occurrences of `raise` and `try...with`, and let the type inference engine figure out the precise type τ_{exn}.

Task 4 Implement a transformation from the full source language to the source language without the `raise` and `try...with` constructs. By running this transformation before compilation, source programs containing `raise` and `try...with` can be executed via the compiler and abstract machine. The file to modify is `exnconv.ml`.

We suggest to use exception-returning style and the corresponding transformation, as described in Leroy's lectures 3 and 4. Try to produce code that is free of administrative redexes. One way to do this is to define the basic operations of the exception monad in such as way that they eliminate administrative redexes "on the fly": see the definition of `bind` in the provided file `exnconv.ml` for an example.

If time allows, try to ensure that your transformation preserves types: if the source program is well-typed, the transformed program should be well-typed also. It is instructive to re-run type inference after translation and compare the inferred types before and after translation.

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

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.
- Reading your source code, for correctness and elegance.

A correct implementation of tasks $1 + 2$ or tasks $1 + 3$ corresponds to a passing grade (10/20).

6 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.

7 Deadline

Please turn in your assignment on or before **Sunday, 24 February 2008**.