MPRI 2.4
Programmation fonctionnelle et systèmes de types
Programming project

François Pottier

2020–2021

Abstract

The purpose of this programming project is to implement a type-checker and an interpreter for a tiny programming language equipped with a simple form of effect handlers.

1 Presentation

System Ξ The purpose of this programming project is to implement a type-checker and an interpreter for System Ξ, a tiny calculus equipped with a form of effect handlers.[1] In short, System Ξ can be described as a programming language that is equipped with effect handlers and that is subject to two important restrictions:

- When a new effect is declared, a handler for this effect is installed at the same time. A capability to perform this effect (that is, a function that performs this effect) is made available to the user.
- Functions are not first-class values. A function can be passed as an argument to a function, but cannot be returned out of a function or stored in the heap.

As a result of these restrictions, a function that performs an effect cannot outlive the handler for this effect. Thus, there is no danger of unhandled effects. This means that an ordinary type system suffices to guarantee a strong type safety result. There is no need for a more complex "type-and-effect system" that would keep track of which function may raise which effect.

A reader who is not familiar with effects can think of them as a more powerful form of exceptions.[2] Performing an effect is analogous to raising an exception. The difference between exception handlers and effect handlers is that an effect handler receives a continuation that allows it to possibly resume the suspended computation (once, more than once, or never).

Type-checking System Ξ System Ξ is a standard λ-calculus where a syntactic distinction between "expressions" and "blocks" (that is, functions) has been introduced. Accordingly, there are two separate syntactic categories of ordinary variables x and block variables F. A block can expect several arguments,

[1]System Ξ is described in Section 4 of the paper "Effekt: Lightweight Effect Polymorphism for Handlers", by Brachthäuser *et al.* [1]. In that paper, the authors propose the programming language Effekt, and present a type-directed translation of Effekt into System Ξ, described in Section 5 of the paper. Understanding these aspects is not required. **We recommend reading Sections 1 and 2** of the paper (no more) for an introduction to Effekt.

[2]One may remark that, in a programming language with exceptions, imposing the two restrictions described above eliminates the danger of uncaught exceptions. Zhang *et al.* make such a proposal [5]. This is not required reading.

which can be either ordinary values or blocks. A block returns just one result, which must be an ordinary value; it cannot be a block. This enforces the restriction that a function cannot be returned out of a function.

One purpose of the type system is to enforce the distinction between values and blocks: for instance, if a block F expects a block as an argument, then F cannot be applied to a value, and vice-versa. The other purpose of the type system is to rule out all runtime errors, as usual. We use a system of simple types (that is, monomorphic types), and we perform type inference. Our type inference algorithm produces unification constraints and relies on the library `inferno` to solve them.

The abstract syntax of System Ξ appears in the file `src/SystemXi.ml`. It also appears in Figure 3a of Brachthäuser *et al.*'s paper [1]. (Our syntax differs slightly from theirs, though.) Their typing rules appear in Figure 3b.

We formulate our typing rules in a slightly different way: instead of maintaining two separate environments for value variables and block variables, we maintain a single environment, and use types of the form "`value` τ" and "`block` τ" to keep track of the distinction. This information also appears in the type of a block: for instance, the type of a block that expects two values as arguments is of the form $(\texttt{value}\ \tau_0\ \texttt{::}\ \texttt{value}\ \tau_1\ \texttt{::}\ \texttt{nil}) \rightarrow \tau$.

Aside from this unusual aspect, our type discipline is essentially standard. The typing rule for `handle` can be taken from Figure 3b of Brachthäuser *et al.*'s paper [1].

Translating System Ξ down to DPJS In order to execute a program expressed in System Ξ, we first translate it to a simpler calculus, which we name DPJS, after Dybvig, Peyton Jones, and Sabry [2]. It is a λ-calculus equipped with four delimited control operators, namely *newPrompt*, *pushPrompt*, *withSubCont*, and *pushSubCont*. **We recommend reading Section 2.1** of their paper, which presents the syntax of the calculus and an example, **and studying Section 3.3**, which presents an operational semantics. It is important to understand this semantics, because it explains what the control operators do, and because it helps understand how we implement the delimited continuation monad.

Translating System Ξ into DPJS is mostly straightforward. A block that takes n arguments is translated to a function that takes an n-tuple as an argument. The main aspect of interest is the translation of System Ξ's `handle` construct into the lower-level (and more expressive) control operators of DPJS. One could say that `handle` is macro-expanded into lower-level constructs.

Interpreting DPJS In order to execute a program expressed in DPJS, we write an interpreter. Following a tradition established by Wadler [4], it is a monadic interpreter: that is, its main function, `eval`, maps an expression (under a certain runtime environment) to a computation in a well-chosen monad.

We use a variant of the delimited continuation monad proposed by Dybvig, Peyton Jones, and Sabry [2]. **We suggest reading Section 6.1** and possibly Section 6.2 of their paper. Our API, found in the file `src/DelimitedContinuationMonad.mli`, is essentially the same as theirs, except that our operation `run` has a more general type than their `runCC`. We do not need the more complicated types that appear in Section 6.3.

Because the difficult work of implementing the control operators is performed inside the monad, the implementation of the interpreter is straightforward.

The delimited continuation monad The last piece in the puzzle is an implementation of the delimited continuation monad. We follow the ideas found in Sections 7.1, 7.2 and 7.3 of Dybvig *et al.*'s paper, with a few improvements. You may read these sections, but that should not be necessary: the indications that we provide should be sufficient.

We provide an implementation of prompts, also known as tags, in the files `src/Tag.ml[i]`. Thus, the bulk of the work is to implement the type of computations, `'a m`, and the operations on this type. The idea is to represent a computation, in continuation-passing style, as a function of an evaluation context to an answer. Furthermore, taking inspiration from the operational semantics of Section 3.3 of Dybvig *et al.*'s paper [2], an evaluation context E is represented as a linked list of frames D and prompts p.

Tasks Although we have presented the various layers from top to bottom, we suggest implementing these layers from bottom to top, beginning with the implementation of the delimited continuation monad and working up towards the surface language. The sources that we provide and your tasks are described next.

2 Overview of the sources and tasks

The directory `src/` contains the following source files:

`Eq.ml` This file defines the equality GADT.

`Tag.ml[i]` These files provides an abstract type of runtime tags. There is an operation that creates a fresh tag, and an operation that tests two tags for equality. When a tag is created, it becomes associated (once and for all) with a certain type `'a`, which the user chooses (and which OCaml infers). The type of this tag is then `'a tag`. When two tags of types `'a tag` and `'b tag` are compared and are found to be equal, one can deduce that the types `'a` and `'b` must be equal. The operation `equal` produces a witness of this type equality, which the type-checker can exploit.

`DelimitedContinuationMonad.ml[i]` These files describe the API of the delimited continuation monad and provide an implementation of this monad. However, this implementation is in fact missing! **It is up to you to provide this implementation (task 1).** We provide quite a few hints in the comments. Furthermore, during your first attempt, there are two simplifications that you can make. First, you can temporarily ignore the invariant that the stack must not contain two consecutive `SFrame` constructors. (Indeed, this invariant helps improve performance, but is not required for correctness.) Second, you can use a fixed answer type (say, `unit`) instead of requiring every computation to be polymorphic in the answer type. This will force you to give a monomorphic type to `run`, but otherwise should not create any problem. Once you have successfully implemented the monad with these simplifications, remove them one by one.

`DPJS.ml` This file defines the abstract syntax of the calculus DPJS.

`Interpret.ml[i]` These files provide an interpreter for the calculus DPJS. Its implementation relies on the delimited continuation monad. However, this implementation is incomplete! **It is up to you to complete it (task 2).**

`SystemXi.ml` This file defines the abstract syntax of System Ξ.

`Translate.ml[i]` These files provide a translation of System Ξ into DPJS. However, the implementation of this translation is missing! **It is up to you to provide it (task 3).**

`TypeCheck.ml[i]` These files provide a monomorphic type inference system for System Ξ. However, its implementation is missing! **It is up to you to provide it (task 4).** We wish to use the following type algebra, which is organized in three separate syntactic categories, or sorts:

$$\begin{array}{lcll} \tau & ::= & \texttt{bool} \mid \texttt{string} \mid \rho \to \tau & \text{ordinary types} \\ \rho & ::= & \alpha \texttt{ :: } \rho \mid \texttt{nil} & \text{rows of argument types} \\ \alpha & ::= & \texttt{value}\ \tau \mid \texttt{block}\ \tau & \text{argument types} \end{array}$$

That is, we have n-ary function types: the domain of the function type constructor is a row, that is, a list of argument types. An argument type is either `value` τ or `block` τ, so as to reflect whether this argument is a value or a block. Thus, attempting to a pass a value as an argument to a function that expects a block, or vice-versa, causes a type error. A type environment maps variables (that is, both value variables and block variables) to argument types.

In the beginning, we suggest restricting your attention to functions of one (value) argument. This allows you to work with a simpler syntax of types: $\tau ::=$ `bool` $\mid$ `string` $\mid \tau \rightarrow \tau$. Once type inference works in this simplified setting, move to the more general setting described above.

To perform type inference, we use Inferno, whose API is described in the file `SolverHi.mli` and is also explained in a paper by Pottier [3]. (The paper explains several features that we do not need here, though, including Hindley-Milner polymorphism and elaboration to System F.) For our purposes, it is sufficient to build constraints that involve equality constraints, conjunctions, and existential quantification. The combinators `def` and `instance_` should be exploited to indicate where the type environment must be extended and looked up. Inferno provides a constraint solver.

`Lexer.mll, LexerUtil.ml[i], Parser.mly` These files define a lexer and parser for System Ξ.

`Main.ml` This file defines the main program. Typing "`make`" in the root directory compiles it and produces the executable program `_build/default/src/Main.exe`. This program expects to receive on its command line the name of a System Ξ source file, such as `test/inputs/hello.th`. It parses this source file, type-checks it (if the command line option `--typecheck` is given), and executes it (if the command line option `--execute` is given).

Testing The directory `test/inputs` contains a number of small programs written in System Ξ. These programs can be given as arguments to the executable program `_build/default/src/Main.exe`, together with the options `--execute`, `--typecheck`, or both.

In order to execute a single source file, say `test/inputs/hello.th`, run the following command, in the project's root directory:

```
make execute src=test/inputs/hello.th
```

In order to typecheck a single source file, use the following command:

```
make typecheck src=test/inputs/hello.th
```

In order to run the entire test suite, run "`make test`" in the root directory. There are separate tests for the interpreter and for the type-checker. For each System Ξ source file, such as `test/inputs/hello.th`, the interpreter and the type-checker are separately invoked. During each of these tests, the standard output and error channel are recorded. This yields four files, named `hello.{execute,typecheck}.{out,err}` and stored in the directory `_build/default/test/inputs`. The command "`make test`" compares the files `hello.execute.out` and `hello.typecheck.out` against the expected-output files `hello.execute.exp` and `hello.typecheck.exp`, which are stored in the directory `test/inputs`. If there are differences, they are highlighted in color, making failures easy to spot. The files `hello.*.err` are not compared against an expected-output file.

A list of all tests can be found in the file `test/inputs/dune.auto`, which is generated by the OCaml script `test/Test.ml`. After adding a new source file in the directory `test/inputs`, or after modifying `test/Test.ml`, run "`make depend`" in the root directory to regenerate `test/inputs/dune.auto`.

The command "`make promote`" overwrites every `.exp` file with the corresponding `.out` file. It is useful when some expected-output files are incorrect or are missing. It can be used, in particular, after a new source file has been added in the directory `test/inputs`, to create a new expected-output file.

3 Required software

To use the sources that we provide, you need OCaml and Menhir. Any reasonably recent version should do. You also need the packages `pprint` and `inferno`. Assuming that you have installed OCaml via `opam`, please issue the following commands:

```
opam update
opam install menhir pprint inferno.20201104
```

4 Evaluation

Tasks 1, 2, and 3 are regarded as mandatory. Task 1 is conceptually rather complex, but requires a small amount of code, and the types should provide strong guidance. Tasks 2 and 3 are conceptually easier. Task 4 is regarded as optional. The necessity of understanding Inferno and the somewhat complex organization of types into three sorts make it a little challenging.

Assignments will be evaluated by a combination of:

- **Testing**. Your compiler will be tested with the input programs that we provide (make sure that "`make test`" succeeds!) and with additional input programs.
- **Reading**. We will browse through your source code and evaluate its correctness and elegance.

5 What to send

When you are done, please send to François Pottier a `.tar.gz` archive containing your completed programming project.

6 Deadline

Please send your project on or before **January 27, 2021**.

References

[1] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effekt: Lightweight effect polymorphism for handlers. To appear at OOPSLA 2020, November 2020.

[2] R. Kent Dybvig, Simon L. Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(6):687–730, 2007.

[3] François Pottier. Hindley-Milner elaboration in applicative style. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, September 2014.

[4] Philip Wadler. The essence of functional programming. In *Principles of Programming Languages (POPL)*, pages 1–14, 1992. Invited talk.

[5] Yizhou Zhang, Guido Salvaneschi, Quinn Beightol, Barbara Liskov, and Andrew C. Myers. Accepting blame for safe tunneled exceptions. In *Programming Language Design and Implementation (PLDI)*, pages 281–295, June 2016.