Programming languages - final project

Konrad Werbliński - 291878

January 7, 2020

1 Introduction

Algebraic effects have received lot of recognition in the functional programming community in recent years. They are a novel approach for handling side effects. Effectful computations were always considered problematic by programming language researchers because they make programs harder to formally describe and reason about. Moreover, restricting side effects helps to write more maintainable and easier to test programs. With the growing popularity of the Haskell language, monads became preferred way of expressing side effects in a pure way. However, they do not compose very well. Algebraic effects can be viewed as resumable exceptions, providing more composable approach for handling side effects than monads.

We introduce Kwoka - stripped down and simplified version of the Koka language [1]. In this report we describe our work on implementation of the type inference for algebraic effects introduced in the article by Daan Leijen [2]. We follow his approach of using extensible rows with scoped labels for tracking effects of computations.

In this report we specify our variant of a language. We describe the details of the implemented type system. Then, we give an overview of the implementation and instructions for building and running Kwoka. Finally we discuss further work on this project.

2 Kwoka language description

Kwoka syntax is based on the syntax of the original Koka language and Rust. It is designed to resemble popular imperative languages, thus providing familiarity for programmers that don't know the functional paradigm. To articulate the familiarity even more and simplify working with effects, multi-argument functions are defined in the uncurried form.

```
fn fib(n)
{
  if n == 0 || n == 1 then
    n
  else
    fib(n - 1) + fib(n - 2)
}
```

Kwoka supports higher order functions and let-polymorphism (generalization occurs for let expressions, but also for top level function definitions).

```
fn compose(f, g)
{
  \x => f(g(x))
}
```

Algebraic effects

The key feature of the Kwoka language are algebraic effects. Effects are defined as the top level definitions. In the definition we specify possible actions for that effect. For simplicity types of actions are not polymorphic. For handling the effects we the use `handle` expression. Inside the `handle` expression programmer has to provide ways of handling all of the effect's actions and handling of the return value of the effectful computation.

```
effect Hello
{
  Hello() :: String
}
```

```
fn makeGreeting(name)
{
  Hello() ^ " " ^ name
}

fn main()
{
  handle<Hello>(makeGreeting("General Kenobi."))
  {
    return(x) => (),
    Hello() => resume("Hello there!")
  }
}
```

Popular example of usage of the algebraic effects is using them to express exceptions. Thus, they don't have to be part of the language anymore.

```
effect Exc
{
  Raise(String)
}

fn saveDiv(n, m)
{
  if m == 0 then
    Raise("Division by zero")
  else
    n / m
}

fn main()
{
  handle<Exc>(saveDiv(4, 5))
  {
    return(x) => x,
    Raise(s) => 0
  }
}
```

Definition scope

Top level functions are visible to themselves and functions below them. Effect definitions are visible in the entire source file.

3 Type Inference

Kwoka uses the Hindley-Milner type system [3, 4] extended for handling inference of the effects. Effects are represented as extensible rows with scoped labels. This concept of extensible rows was originally proposed for implementing extensible records with scoped labels [5], however they fit very well for tracking the effects of computations.

$$\frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma \mid \epsilon} \qquad \frac{\Gamma \vdash e_1 : \sigma \mid \epsilon \qquad \Gamma, x : \sigma \vdash e_2 : \tau \mid \epsilon}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau \mid \epsilon} \qquad \frac{\Gamma \vdash e : \tau \mid \langle\rangle \qquad \overline{\alpha} \notin \text{ftv}(\Gamma)}{\Gamma \vdash e : \forall \overline{\alpha}.\tau \mid \epsilon} \qquad \frac{\Gamma \vdash e : \forall \overline{\alpha}.\tau \mid \epsilon}{\Gamma \vdash e : \tau[\overline{\alpha} \mapsto \overline{\tau}] \mid \epsilon}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \mid \epsilon'}{\Gamma \vdash \lambda x.e : \tau_1 \to \epsilon' \tau_2 \mid \epsilon} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \epsilon\, \tau_2 \mid \epsilon \qquad \Gamma \vdash e_2 : \tau_1 \mid \epsilon}{\Gamma \vdash e_1(e_2) : \tau_2 \mid \epsilon}$$

$$\frac{\Gamma, x : \tau \vdash e_r : \tau_r \mid \epsilon \qquad \Gamma \vdash e : \tau \mid \langle l \mid \epsilon \rangle \qquad \Sigma(l) = \{op_1, \ldots, op_n\} \qquad \Gamma \vdash op_i : \tau_i \to \langle l \rangle \tau_i' \mid \langle\rangle \qquad \Gamma, \mathit{resume} : \tau_i' \to \epsilon\, \tau_r, x_i : \tau_i \vdash e_i : \tau_r \mid \epsilon}{\Gamma \vdash \mathbf{handle}(e)\{op_1(x_1) \to e_1; \ldots; op_n(x_n) \to e_n; \mathbf{return}(x) \to e_r\} : \tau_r \mid \epsilon}$$

We use the unification algorithm described in the Leijen's article about extensible records [5].

TODO Unification rules

4 Implementation

Kwoka is implemented in Haskell, as we believe this language has cleaner syntax and more elegant way of handling effectful computations than other mainstream functional languages.

Source code structure

AST

In the AST module we define types to describe syntax of the expressions, effect definitions and types. Moreover, we provide pretty printing algorithm by making all of the above mentioned data types instances of the `Show` type class.

Parser

For parsing we use the megaparsec library, which is an extended and improved version of the popular parsec library for monadic parsing.

Preliminary

After the parsing phase we move to preliminary phase, where type and effect environments are built from the effect definitions.

Type inference

In the type inference module we implement the Hindley-Milner type inference algorithm extended for handling algebraic effects. We use monad transformers to compose `State` and `Either` monads, used for fresh variable names generating and type error signaling. Aside from that, in this module we define type for representing type errors and we make it an instance of the `Show` type class for converting them into the human readable form.

Main

In the main module we bind the algorithm together, handle reading the command line arguments and handle loading of the source file.

Upon successful parsing and type checking, Kwoka pretty prints the parsed AST to the standard output. In the other case, it prints the human readable representation of the encountered error.

Tests

In the `test` folder we provide over 85 unit tests for the type system.

5 Installing and running Kwoka

Kwoka requires stack for building.

- Building: `stack build`
- Running: `stack run yourSourceFile.kwoka`
- Running unit tests: `stack test`

In the `example` folder we provide a handful of examples of the Kwoka language.

6 Further work

- We plan to implement the evaluation and the type directed selective CPS translation described in the second part of the Daan Leijen's article[2].
- Small and simple, but extremely useful extension would be addition of list and product types.
- Another useful feature would be inclusion of effect actions with polymorphic types.
- We plan to refactor unit tests to use popular testing framework for Haskell: HUnit.

References

[1] Daan Leijen. Koka: Programming with row polymorphic effect types. *Electronic Proceedings in Theoretical Computer Science*, 153, 06 2014.

[2] Daan Leijen. Type directed compilation of row-typed algebraic effects. In *Proceedings of Principles of Programming Languages (POPL'17), Paris, France*, January 2017.

[3] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[4] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[5] Daan Leijen. Extensible records with scoped labels. In *Proceedings of the 2005 Symposium on Trends in Functional Programming (TFP'05), Tallin, Estonia*, September 2005.