Programming languages - final project

Konrad Werbliński - 291878

May 31, 2021

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 [2] and abstract machine [3] for algebraic effects.

In the next section we specify our variant of a language. We describe the details of the implemented type system and abstract machine. 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)
{
  fn (x) => f(g(x))
}
```

The language also features product and list types.

```
fn fst(p)
{
  let (x, y) = p in x
}

fn map(f, xs)
{
  case xs of
    [] => []
    x : xs => f(xs) : map(f, xs)
}
```

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 operations for that effect. For simplicity types of operations are not polymorphic. For handling the effects we use `handle` expression. Inside the `handle` expression programmer has to provide ways of handling all of the effect's operations and handling of the return value of the effectful computation. Popular example of usage of 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()
{
  let y = ReadLnInt() in
  handle<Exc>(saveDiv(400, y))
  {
    return(x) => PrintInt(x),
    Raise(s) => PutStrLn("Division by zero")
  }
}
```

Algebraic effects can be also used for simulating computations with nondeterminism. In the example below we define effect that represents the coin toss. Then, in the handler of that effect we use multiple resumptions to generate all sublists of the list.

```
effect Toss
{
  Coin() :: Bool
}

fn sublist(xs)
{
  case xs of
    [] => []
    x : xs =>
      if Coin() then
        x : sublist(xs)
      else
        sublist(xs)
}

fn main()
{
  let res = handle<Toss>(sublist([1,2,3,4,5]))
  {
    return(x) => [x],
    Coin() => resume(True) @ resume(False)
  } in

  iter (fn (xs) => let () = iter(fn (x) => PrintInt(x) , xs) in PutStrLn(""), res)
}
```

Kwoka provides default definition and handler for the IO effect, which consists of following operations: `GetLine`, `ReadLnInt`, `PutStrLn`, `PrintInt`.

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

Types:

$\tau ::=$

`Bool` | `Int` | `String`	simple types
$\mid (\tau_1, \tau_2, \ldots \tau_n)$	product
$\mid [\tau]$	lists
$\mid \tau_1 \to \langle \epsilon \rangle \tau_2$	arrows
$\mid \alpha$	type variable

Type schemes:

$\sigma ::=$

$\forall \overline{\alpha}.\, \tau$

Effect types

$\epsilon ::=$

$\langle l \mid \epsilon \rangle$	label
$\mid \mu$	effect variable
$\mid \langle\rangle$	empty effect row

Kwoka uses the Hindley-Milner type system [4, 5] extended for algebraic effects. We follow Daan Leijen's [2] approach of using extensible rows with scoped labels for effect tracking. The concept of extensible rows was originally proposed for implementing extensible records with scoped labels [6], however they fit very well for tracking the effects of computations.

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

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

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

We use the unification algorithm described in the Leijen's article [6]. We omit the unification rules for product and list types, since they are straightforward.

$$c ::= \texttt{Bool} \mid \texttt{Int} \mid \texttt{String}$$

$$c \sim c : [] \qquad \alpha \sim \alpha : [] \qquad \frac{\alpha \notin \text{ftv}(\tau)}{\alpha \sim \tau : [\alpha \mapsto \tau]} \qquad \frac{\alpha \notin \text{ftv}(\tau)}{\tau \sim \alpha : [\alpha \mapsto \tau]}$$

$$\frac{\tau_1 \sim \tau_1' : \Theta_1 \qquad \Theta_1\epsilon \sim \Theta_1\epsilon' : \Theta_2 \qquad (\Theta_2 \circ \Theta_1)\tau_2 \sim (\Theta_2 \circ \Theta_1)\tau_2' : \Theta_3}{\tau_1 \to \langle \epsilon \rangle \tau_2 \sim \tau_1' \to \langle \epsilon' \rangle \tau_2' : \Theta_3 \circ \Theta_2 \circ \Theta_1} \qquad \langle\rangle \sim \langle\rangle : [] \qquad \mu \sim \mu : []$$

$$\frac{\mu \notin \text{ftv}(\epsilon)}{\mu \sim \epsilon : [\mu \mapsto \epsilon]} \qquad \frac{\mu \notin \text{ftv}(\epsilon)}{\epsilon \sim \mu : [\mu \mapsto \epsilon]} \qquad \frac{\epsilon_2 \simeq \langle l \mid \epsilon_2' \rangle : \Theta_1 \qquad \text{tail}(\epsilon_1) \notin \text{dom}(\Theta_1) \qquad \Theta_1\epsilon_1 \sim \Theta_1\epsilon_2' : \Theta_2}{\langle l \mid \epsilon_1 \rangle \sim \epsilon_2 : \Theta_2 \circ \Theta_1}$$

$$\langle l \mid \epsilon \rangle \simeq \langle l \mid \epsilon \rangle : [] \qquad \frac{l \neq l' \qquad \epsilon \simeq \langle l \mid \epsilon' \rangle : \Theta}{\langle l' \mid \epsilon \rangle \simeq \langle l \mid l' \mid \epsilon' \rangle : \Theta} \qquad \frac{\text{fresh}(\mu')}{\mu \simeq \langle l \mid \mu' \rangle : [\mu \mapsto \langle l \mid \mu' \rangle]}$$

4 Abstract machine

We implemented slightly modified version of the abstract machine introduced by Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk and Filip Sieczkowski [3].

$$e \Rightarrow \langle e, \{\}, \bullet, \bullet \rangle_{eval}$$

$$\langle x, \rho, \kappa, \pi \rangle_{eval} \Rightarrow \langle \kappa, v, \pi \rangle_{stack} \text{ where } v = \rho(x)$$
$$\langle \lambda x.e, \rho, \kappa, \pi \rangle_{eval} \Rightarrow \langle \kappa, \lambda^{\rho} x.e, \pi \rangle_{stack}$$
$$\langle (), \rho, \kappa, \pi \rangle_{eval} \Rightarrow \langle \kappa, (), \pi \rangle_{stack}$$
$$\langle (e_1, e_2, \ldots, e_n), \rho, \kappa, \pi \rangle_{eval} \Rightarrow \langle e_1, \rho, \{()(e_2, \ldots, e_n)\}^{\rho} : \kappa, \pi \rangle_{eval}$$
$$\langle op_l, \rho, \kappa, \pi \rangle_{eval} \Rightarrow \langle \kappa, op_l, \pi \rangle_{stack}$$
$$\langle e_1 e_2, \rho, \kappa, \pi \rangle_{eval} \Rightarrow \langle e_1, \rho, e_2^{\rho} : \kappa, \pi \rangle_{eval}$$
$$\langle primitive, \rho, \kappa, \pi \rangle_{eval} \Rightarrow \langle \kappa, primitive, \pi \rangle_{stack}$$
$$\langle \mathbf{handle} \langle l \rangle (e) \{hr\}, \rho, \kappa, \pi \rangle_{eval} \Rightarrow \langle e, \rho, \bullet, (hr_l^{\rho}, \kappa) : \pi \rangle_{eval}$$

$$\langle \bullet, v, \pi \rangle_{stack} \Rightarrow \langle \pi, v \rangle_{mstack}$$
$$\langle e^{\rho} : \kappa, v, \pi \rangle_{stack} \Rightarrow \langle e, \rho, v : \kappa, \pi \rangle_{eval}$$
$$\langle \{(v_1, \ldots, v_{n-1})()\}^{\rho} : \kappa, v_n, \pi \rangle_{stack} \Rightarrow \langle \kappa, (v_1, \ldots, v_{n-1}, v_n), \pi \rangle_{stack}$$
$$\langle \{(v_1, \ldots, v_{k-1})(e_{k+1}, e_{k+2}, \ldots, e_n)\}^{\rho} : \kappa, v_k, \pi \rangle_{stack} \Rightarrow \langle e_{k+1}, \rho, \{(v_1, \ldots, v_{k-1}, v_k)(e_{k+2}, \ldots, e_n)\}^{\rho} : \kappa, \pi \rangle_{eval}$$
$$\langle \lambda^{\rho} x.e : \kappa, v, \pi \rangle_{stack} \Rightarrow \langle e, \rho\{x \mapsto v\}, \kappa, \pi \rangle_{eval}$$
$$\langle primitive : \kappa, v, \pi \rangle_{stack} \Rightarrow \langle \kappa, v', \pi \rangle_{stack} \text{ where } v' = [\![primitive\ v]\!]$$
$$\langle op_l : \kappa, v, \pi \rangle_{stack} \Rightarrow \langle op_l, \kappa, \pi, v, \bullet \rangle_{op}$$
$$\langle \theta : \kappa, v, \pi \rangle_{stack} \Rightarrow \langle \theta, \kappa, \pi, v \rangle_{res}$$

$$\langle op_l, \kappa, (\{h; d\}_l^{\rho}, \kappa') : \pi, v, \theta \rangle_{op} \Rightarrow \langle e, \rho\{x \mapsto v\}\{r \mapsto (\{h; d\}_l^{\rho}, \kappa) : \theta\}, \kappa', \pi \rangle_{eval} \text{ where } op\, x,\ r.e \in h$$

$$\langle op_l, \kappa, (hr_{l'}^{\rho}, \kappa') : \pi, v, \theta \rangle_{op} \Rightarrow \langle op_l, \kappa', \pi, v, (hr_{l'}^{\rho}, \kappa) : \theta \rangle_{op} \text{ if } l \neq l'$$

$$\langle \bullet, \kappa, \pi, v \rangle_{res} \Rightarrow \langle \kappa, v, \pi \rangle_{stack}$$
$$\langle (\mu, \kappa') : \theta, \kappa, \pi, v \rangle_{res} \Rightarrow \langle \theta, \kappa', (\mu, \kappa) : \pi, v \rangle_{res}$$

$$\langle (\{h; \mathbf{return}(x).\, e\}_l^{\rho}, \kappa) : \pi, v \rangle_{mstack} \Rightarrow \langle e, \rho\{x \mapsto v\}, \kappa, \pi \rangle_{eval}$$
$$\langle \bullet, v \rangle_{mstack} \Rightarrow v$$

5 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.

- Expressions:

```
data Expr p
  = EVar      p Var
  | EBool     p Bool
  | EInt      p Integer
  | EString   p String
  | ENil      p
  | EUnOp     p UnOp (Expr p)
  | EBinOp    p BinOp (Expr p) (Expr p)
  | ELambda   p [Var] (Expr p)
  | EApp      p (Expr p) (Expr p)
  | EIf       p (Expr p) (Expr p) (Expr p)
  | ELet      p Var (Expr p) (Expr p)
  | EOp       p Var (Expr p)
  | EHandle   p Var (Expr p) [Clause p]
  | ETuple    p [Expr p]
  | ELetTuple p [Var] (Expr p) (Expr p)
  | ECase     p (Expr p) (Expr p) (Var, Var) (Expr p)
```

- Types:

```
data Type
  = TVar TypeVar
  | TBool
  | TInt
  | TString
  | TList Type
  | TProduct [Type]
  | TArrow Type EffectRow Type
  deriving Eq
```

- Type schemes:

```
data TypeScheme = TypeScheme [TypeVar] Type
```

- Effect rows:

```
data EffectRow
  = EffLabel Var EffectRow
  | EffEmpty
  | EffVar TypeVar
  deriving Eq
```

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. In this module we also check the well-formedness of the definitions and define the types of the builtin operators.

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.

- `infer` - main function of the inference algorithm, infers the types of expressions.
- `check` - helper function that infers expression type and then unifies it with a given type.
- `unify` - unification of types.
- `unifyRow` - unification of effect rows.

MachineAST

In the MachineAST module we define types that represent expressions in the abstract machine language. We define primitive operations which represent unary and binary operators and IO operations (used for implementation of the default IO effect handler, which is written in the abstract machine code). In this module we also define data structures necessary for the machine to work, such as stack frames and values.

- Machine expressions:

```
data MExpr
  = MVar     MVar
  | MInt     Integer
  | MBool    Bool
  | MString  String
  | MNil
  | MTuple   [MExpr]
  | MPrim    MPrim MExpr
  | MLambda  [MVar] MExpr
  | MApp     MExpr MExpr
  | MIf      MExpr MExpr MExpr
  | MOp MVar MVar MExpr
  | MCase    MExpr MExpr (MVar, MVar) MExpr
  | MHandle  MVar MExpr (Map.Map MVar MClause)
  deriving (Show, Eq)
```

- Primitive operations:

```
data MPrim
  = Not | Neg | Mult | Div | Mod | Add | Sub
  | Concat | Cons | Append | Equal | NotEqual
  | LessEqual | GreaterEqual | Less | Greater
  | And | Or | Print | GetLine | ReadLnInt deriving (Show, Eq)
```

- Values:

```
data MValue
  = VInt Integer
  | VBool Bool
  | VString String
  | VList [MValue]
  | VTuple [MValue]
  | VClosure RuntimeEnv [MVar] MExpr
  | VReifiedMetaStack MetaStack
```

- Stack frames:

```
data MFrame
  = FArg RuntimeEnv MExpr
  | FClosure RuntimeEnv [MVar] MExpr
  | FOp MVar MVar
  | FPrim MPrim
  | FTuple RuntimeEnv [MValue] [MExpr]
  | FIf RuntimeEnv MExpr MExpr
  | FCase RuntimeEnv MExpr (MVar, MVar) MExpr
  | FReifiedMetaStack MetaStack
  deriving (Show, Eq)
```

- Meta-stack frames:

```
data MMetaFrame
  = MMetaFrame (MVar, Map.Map MVar MClause, RuntimeEnv) Stack
  deriving (Show, Eq)
```

Translate

In this module we implement translation algorithm between abstract syntax and machine code. During this translation we also perform basic constant folding.

- `translate` - translates expressions to machine expressions.
- `binOpToMPrim` - translates binary operators to primitive operations.
- `ioHandler` - implementation of the default IO handler, which is placed over the main function.

Machine

In this module we define five functions which correspond to five configurations of the abstract machine. We use the IO monad to implement primitive operations for input-output.

- `eval :: MExpr -> RuntimeEnv -> Stack -> MetaStack -> IO MValue`
- `stack :: Stack -> MValue -> MetaStack -> IO MValue`
- `op :: (MVar, MVar) -> Stack -> MetaStack -> MValue -> MetaStack -> IO MValue`
- `resume :: MetaStack -> Stack -> MetaStack -> MValue -> IO MValue`
- `mstack :: MetaStack -> MValue -> IO MValue`

Main

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

Tests

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

6 Installing and running Kwoka

Kwoka requires stack for building.

- Building: `stack build`.
- Running: `stack run -- yourSourceFile.kwoka [-debug]`.
- Adding `-debug` flag pretty-prints typed typed program to a file.
- Running unit tests: `stack test`.
- To install kwoka on your machine: `stack install`, then you can run kwoka in your terminal by running `kwoka-exe`.

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

7 Further work

- We plan to implement the type directed selective CPS translation [2] as a next step towards efficient programming language implementation.
- Useful feature would be inclusion of effect operations 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] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Handle with care: relational interpretation of algebraic effects and handlers. *Proceedings of the ACM on Programming Languages*, 2:1–30, 12 2017.

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

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

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