A new polymorphic type system for Miking

Anders Ågren Thuné (athune@kth.se)

Miking Workshop 2023

2023-11-23

Type system key contributors (alphabetical order):
David Broman, Elias Castegren, Viktor Palmkvist, Anders Ågren Thuné

Outline

- Background
- Limitations of the current type checker
- The new features
- Implementation
- Future challenges

Types in Miking

- Miking was originally developed without a type checker.
- Can you spot the bug?

```
sem eval env =
| TmLet t ->
  eval
    (insert t.ident (eval t.body) env)
    t.inexpr
```

```
let x = 5 in addi x 1
```

Types in Miking

- Miking was originally developed without a type checker.
- Can you spot the bug?

```
sem eval env =
| TmLet t ->
  eval
    (insert t.ident (eval t.body) env)
    t.inexpr
```

```
let x = 5 in addi x 1
```

- Without type check: runtime error

Types in Miking

- Miking was originally developed without a type checker.
- Can you spot the bug?

```
sem eval env =
| TmLet t ->
  eval
    (insert t.ident (eval env t.body) env)
    t.inexpr
```

```
let x = 5 in addi x 1
```

- Without type check: runtime error

Type Checking

- We want to catch the “obvious” errors *before* runtime.

```
sem eval : Env -> Expr -> Value
```

Type Checking

- We want to catch the “obvious” errors *before* runtime.

```
sem eval : Env -> Expr -> Value
```

```
ERROR </path/to/eval.mc 124:53-124:59>:
* Expected an expression of type: Env
*    Found an expression of type: Expr
  ...
* When type checking the expression
    (insert t.ident (eval t.body) env)
```

Type Checking

- We want to catch the “obvious” errors *before* runtime.

```
sem eval : Env -> Expr -> Value
```

```
ERROR </path/to/eval.mc 124:53-124:59>:
* Expected an expression of type: Env
*    Found an expression of type: Expr
  ...
* When type checking the expression
    (insert t.ident (eval t.body) env)
```

- Development started in September 2021 and type checking was enabled for `mi` universally in June 2022. Based on FreezeML [1] for first-class polymorphism.

[1] Emrich et al., PLDI ’20

Limitations of the Type Checker

- Is this code okay?

```
type Expr
con  TmInt : Int -> Expr
let getInt = lam x.
  match x with TmInt i then i
  else never

getInt (TmInt 0)
```

Limitations of the Type Checker

- Is this code okay?

```
type Expr
con  TmInt : Int -> Expr
let getInt = lam x.
  match x with TmInt i then i
  else never

getInt (TmInt 0)
```

- What about now?

```
...
con TmString : String -> Expr
getInt (TmString "hello")
```

Constructor Types

- We annotate the type with a set of constructors

```
getInt : Expr{TmInt} -> Int
```

Constructor Types

- We annotate the type with a set of constructors

```
getInt : Expr{TmInt} -> Int
```

- Now we can differentiate different versions of a type

```
getInt (TmInt 0)           -- Ok!
getInt (TmString "hello")  -- Error: Expected Expr{TmInt}, found Expr{TmString}
```

Open Types

- What if any constructors are okay?

```
type Foo
con  F1 : Int -> Foo
con  F2 : Foo -> Foo

let f = lam x.
  match x with F2 _ then x
  else F1 0
```

Open Types

- What if any constructors are okay?

```
type Foo
con  F1 : Int -> Foo
con  F2 : Foo -> Foo

let f = lam x.
  match x with F2 _ then x
  else F1 0
```

- We use polymorphism to express *open types*

```
f : Foo{a} -> Foo{a}
    where a :: {F1,F2}
```

A New Formalism

- A new formalization of Miking's core system

$$
\begin{array}{lcll}
\tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \forall\alpha :: \kappa.\ \tau \mid \tau_1 \times \tau_2 \mid \tau_1 \rightsquigarrow^{\overline{p}} \tau_2 \mid \tau.T \mid \delta & \text{(Types)} \\
\kappa & ::= & \star \mid \delta & \text{(Kinds)} \\
\delta & ::= & \langle T_1 : \overline{\mathsf{K}}_1, \ldots, T_n : \overline{\mathsf{K}}_n \rangle & \text{(Constructor types)} \\
e & ::= & x \mid \lambda x : \tau.\ e \mid e_1\ e_2 \mid \Lambda\alpha :: \kappa.\ e \mid e[\tau] & \text{(Expressions)} \\
 & \mid & \mathbf{fix}\lambda^{\tau_1 \to \tau_2}\ f\ x.\ e \mid (e_1, e_2) \mid \pi_1\ e \mid \pi_2\ e & \\
 & \mid & \mathsf{K}\ [\tau]\ e \mid \mathbf{match}\ e\ \mathbf{with}\ p\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid \mathbf{never}^{\tau} & \\
 & \mid & \mathbf{type}\ T\ \mathbf{in}\ e \mid \mathbf{con}\ \mathsf{K} : (X :: \delta).\ \tau \to T\ \mathbf{in}\ e & \\
 & \mid & \mathbf{sem}^{\tau}\ \{p \to e\} \mid e_1 \oplus e_2 \mid e_1 \bullet e_2 & \\
\Gamma & ::= & \cdot \mid \Gamma, x : \tau \mid \Gamma, \alpha :: \kappa \mid \Gamma, e \triangleright p \mid \Gamma, e \triangledown p & \text{(Typing environments)} \\
 & \mid & \Gamma, T \mid \Gamma, \mathsf{K} : (X :: \delta).\ \tau \to T &
\end{array}
$$

Implementation in Miking

- Extension with constructor types and a new exhaustiveness checker

```
sem typeCheckExpr env =
| TmNever t ->
  match matchesPossible env with None () then
    TmNever {t with ty = newpolyvar env.currentLvl t.info}
  else ...
    errorSingle [t.info] msg
```

Implementation in Miking

- Extension with constructor types and a new exhaustiveness checker

```
sem typeCheckExpr env =
| TmNever t ->
  match matchesPossible env with None () then
    TmNever {t with ty = newpolyvar env.currentLvl t.info}
  else ...
    errorSingle [t.info] msg
```

- Language fragments for extensible and reusable code

```
lang AppTypeCheck = TypeCheck + AppAst
  sem typeCheckExpr env =
  | TmApp t -> ...
end ...
lang MExprTypeCheck = LamTypeCheck + AppTypeCheck + ... + MExprPatAnalysis + ...
```

Future Work

- Better error messages and editor support.

Future Work

- Better error messages and editor support.
- Expanded features (GADTs, gradual typing, ...) and formalization.

Future Work

- Better error messages and editor support.
- Expanded features (GADTs, gradual typing, ...) and formalization.
- Error messages at the level of MLang.

Summary

- The ML-style type checker offers easy and expressive typing for MExpr. You can try it out today!
- Constructor types and exhaustiveness checker coming soon^{TM}.

- Outlook
  - Better programmer support.
  - Extended features.