An Introduction to the Type System of MCore

Miking Workshop 2025

Anders Ågren Thuné

Type Checking MCore

- The current type checker was introduced in 2021 and enabled by default in 2022.
- Implements a **lightweight** but **powerful** type system with **type inference**.

> ⓘ **Note**
>
> Currently, only MExpr code is type checked.
>
> MLang is type checked after translation to MExpr.

Catching “Obvious” Bugs

- Can you spot the bug?

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

Catching "Obvious" Bugs

- Can you spot the bug?

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

The Zoo of MCore Types

Basic Types

```
let count    : Int     = 5
let shoeSize : Float   = 45.9
let letter   : Char    = 'a'
let isLetter : Bool    = isAlpha letter
let foo      : Unknown = count
```

Function Types

```
let idInt : Int -> Int = lam x. x

let applyn : Int -> (Int -> Int) -> (Int -> Int) =
  lam n. lam f.
  match n with 0 then
    f
  else
    lam x. applyn (subi n 1) f (f x)
```

Sequences

```
let fib10 : [Int]  = [1,1,2,3,5,8,13,21,34,55]
let name  : [Char] = "Batman"
let name2 : String = name
```

Tuples and Records

```
type Point = (Float, Float)

let subpt : Point -> Point -> Point = lam p1. lam p2.
  (subf p1.0 p2.0, subf p1.1 p2.1)

let person : {name : String, age : Int} =
  {name = "Spider Man", age = 28}
```

> ⓘ **Note**
>
> - Tuples are records: `Point = {#label"0" : Float, #label"1" : Float}`
> - Field order doesn't matter

Algebraic Data Types (syn)

```
lang Regex
  syn Exp =
  | RNull  ()                 -- Ø
  | REmpty ()                 -- ε
  | RChar Char                -- a
  | ROr {e1 : Exp, e2 : Exp} -- e1 | e2

  sem deriv : Char -> Exp -> Exp
  sem deriv a =
  | RChar b -> if eqc a b then REmpty () else RNull ()
  | ROr es  -> ROr {e1 = deriv a es.e1, e2 = deriv a es.e2}
end
```

Parametric Polymorphism

```
let id : all a. a -> a =
  lam x. x

let applyn : all a. Int -> (a -> a) -> (a -> a) =
  lam n. lam f.
  match n with 0 then
    id
  else
    lam x. applyn (subi n 1) f (f x)
```

Advanced Types

First-class Polymorphism

```
let applyn : all a. Int -> (all b. b -> b) -> (a -> a) =
  lam n. lam f.
  match n with 0 then
    id
  else
    lam x. applyn (subi n 1) #frozen"f" (f x)

let _ex : Int = applyn 5 #frozen"id" 2
```

[1]

[1] Based on FreezeML (Emrich et al., PLDI '20)

Record Polymorphism

```
let getName : all a. all b::{name : a}. b -> a = -- syntax does not exist :)
  lam x. x.name
```

[1]

[1] Similar to Ohori’s polymorphic records (Ohori 1995)

Constructor Polymorphism

```
let getChar : all a::{Exp[< RChar]}. Exp{a} -> Char =
  lam re. match re with RChar c then c else never

let getCharOpt : all a::{Exp[> ]}. Exp{a} -> Char =
  lam re. match re with RChar c then c else 'a'
```

[1]

[1] Similar to polymorphic variants (Garrigue 1998)

Future Plans

Typed Language Contraction/Extension

- We want a type system at the level of MLang which can manage different versions of the same datatype from different language fragments.

```
lang Sugar = Expr
  syn Expr =
  | TmLet {ident : Name, body : Expr, inexpr : Expr}

  sem Desugar : Sugar.Expr -> Expr.Expr =
  | TmLet {ident = x, body = t1, inexpr = t2} ->
    TmApp {lhs = TmAbs {ident = x, body = t1}, rhs = t2}
end
```

Generalized Algebraic Datatypes

- Generalized algebraic datatypes (GADTs) give more precise and expressive types

```
lang Effect
  syn Eff a =
  | Pure a
  | Impure (b, b -> Eff a)
end
```

Composable Effects

Different language fragments work in different contexts and produce different effects

- Type checking / evaluation environments
- Mutable state
- Error handling

We don’t want every language fragment to account for every effect explicitly.

Wrapping Up

(Re-)using the Type Checker

- The type checker is implemented as reusable Miking language fragments

```
lang MExprTypeCheck =
  AppTypeCheck + MatchTypeCheck + ConstTypeCheck + SeqTypeCheck +
  RecordTypeCheck + TypeTypeCheck + DataTypeCheck + UtestTypeCheck +
  NeverTypeCheck + ExtTypeCheck + PlaceholderTypeCheck + DeclTypeCheck +
  ...
```

Summary

- Type checking finds bugs!
- MCore features a **lightweight** type system with **type inference**.
- Similar to languages like OCaml or Haskell, but with a focus on extension and composition.
- More interesting features planned in the future.
- Try it yourself!