TinyML: Tiny functional language interpreter

Interpreter and step-by-step guide

Tomas Petricek, Charles University

tomas@tomasp.net
@tomasp.net
https://tomasp.net
https://d3s.mff.cuni.cz/teaching/nprg077

```
type Expression =
  | Constant of int
  | Binary of
      string *
      Expression *
      Expression

val evaluate :
  Expression -> int
```

Basic interpreter structure (0/2)

`Expression` is the source code that user writes

`evaluate` takes expression and returns the result

Charles University

```
type Value =
  | Number of int

type Expression =
  | Constant of int
  | Binary of
      string *
      Expression *
      Expression

val evaluate :
  Expression -> Value
```

Basic interpreter structure (1/2)

Adding values as the result of evaluation

`Value` is what we get as the result

`evaluate` takes expression and returns value

Basic interpreter structure (2/2)

```
type Value =
  | Number of int

type Expression =
  | Constant of int
  | Binary of
      string *
      Expression *
      Expression
  | Variable of string

type VariableContext =
  Map<string, Value>

val evaluate :
  Expression -> VariableContext -> Value
```

Adding variables and variable context

Variable can store only values (call-by-value)

`evaluate` takes context

Demo

Adding values and variables

Lab overview

TinyML interpreter step-by-step

TinyML - Basic tasks

1. Simple numerical evaluator as the starting point
   This has already been done for you :-)
2. Add unary operators (-) and conditional
   We only have numbers, so treat `1` as `true`
3. Functions and application
   Tricky! Closure needs to capture variables!
4. Let binding as syntactic sugar
   Evaluate `let` by treating it as apply/lambda
5. Add a simple data type - tuples
   New value, constructor and destructor

TinyML - Bonus & super tasks

1. Add more data types - unions
   New value, constructor and destructor (match)
2. Add support for recursion
   Needs `Lazy<Value>` in variable context to work
3. Add unit and create a list value
   `Case1(Const(1), Case1(Const(2), Case2(Unit)))`
4. Implement call-by-name semantics
   Change variable context to store expressions
5. Implement evaluation by substitution
   Toy approach, but you learn the semantics

Charles University

Lessons learned

Functional language interpreter

 Distinguishing `Value` and `Expression`

 Recursive function with variable scope

 Call-by-value and lexical variable scoping!

 Nice constructor and destructor symmetry