NPRG077

TinyML: A tiny functional programming language interpreter

Tomáš Petříček, 309 (3rd floor)

✉ petricek@d3s.mff.cuni.cz

➜ https://tomasp.net | @tomaspetricek

➜ https://d3s.mff.cuni.cz/teaching/nprg077

Two sides of this lecture

Introducing the F# language

- Some practical information
- Enough so that you can use it!
- Some important things omitted

Welche Thiere gleichen einander am meiſten?

Kaninchen und Ente.

Introducing ML-style languages

- Background for our TinyML
- Basic features & principles
- Some weird corner cases!

The F# language

What you need to know

Charles University

Getting started with F#

F# and .NET runtime

- .NET SDK for Mac, Linux, Windows
- OSS with .NET Foundation since 2017
- https://dotnet.microsoft.com

F# editors and tools

- Microsoft Visual Studio (Win only)
- JetBrains Rider (Win, Linux, Mac)
- VS Code with Ionide (Win, Linux, Mac)
- https://ionide.io

Charles University

F# project types

Script-based development

- Write code in `.fsx` file
- Run using F# Interactive REPL
- Can select & run out-of-order!

Project-based development

- Project `.fsproj` with `.fs` sources
- Standard build and run workflow
- Live reload with Fable and JavaScript

Charles University

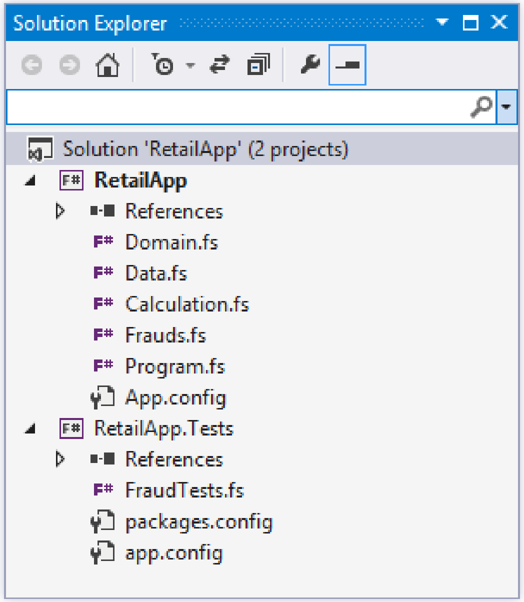
F# project structure

Declaration order matters!

- Helper function and types
- Types defining domain model
- Operations for working with it
- User interface

How to organize F# projects

- Namespaces or modules
- Type declarations
- Functions (inside modules)



Solution 'RetailApp' (2 projects)
- RetailApp
  - References
  - Domain.fs
  - Data.fs
  - Calculation.fs
  - Frauds.fs
  - Program.fs
  - App.config
- RetailApp.Tests
  - References
  - FraudTests.fs
  - packages.config
  - app.config

Data type declarations in F#

- Tuples and records
  Store multiple values of different types
- Discriminated unions
  Represent one of multiple possible options
- Collections, lists and maps
  Multiple values of the same type
- Recursive declarations
  Type that can include values of itself
- Type aliases
  Shorthand for a type with a long name

Charles University

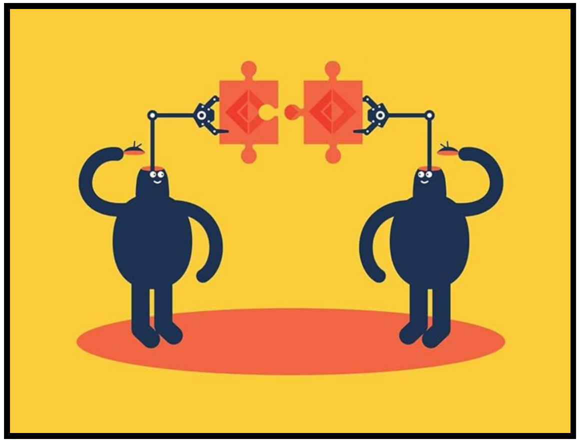
Demo

Simple expression evaluator

Selected advanced features

Lists and maps

- Immutable collections
- Linked (cons) lists with head/tail
- Key-value maps with lookup



Recursion and laziness

- Recursive functions using `let rec`
- Works also for values, but beware!
- `Lazy<T>` to represent lazy values

Charles University

Demo

Maps, lazy values, recursion

TinyML

Interpreter structure

Charles University

Basic interpreter structure (1/2)

```
type Value =
  | Number of int

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

val evaluate :
  Expression -> Value
```

`Expression` is the source code that user writes

`Value` is what we get as the result

`evaluate` takes expression and returns value

Basic interpreter structure (2/2)

```
type Value =
  | ValNum 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

TinyML

How ML languages work

Charles University

Language features of TinyML (1/2)

```
(* Functions *)
let f = (fun x -> 10 + x)
f 32

(* Tuples *)
let t = (1, "hi")
fst t
snd t

(* Unions *)
let c1 = Case1(10)
let c2 = Case2(32)
match c1 with
| Case1 n -> n + 32
| Case2 n -> n + 10
```

Functions but only with single argument

Tuples of two element with getters

Unions without tag name with two cases

Charles University

Language features of TinyML (2/2)

```
(* Let bindings *)
let x = 10 in x * 32

(* Let desugaring *)
(fun x -> x * 32) 10

(* Conditionals *)
if e then 10 else 32

(* Both are expressions *)
1 + (if e then 41 else 1)
1 + (let x = 1 in x + x)

(* Currying *)
let add = fun a -> fun b -> a + b
in (add 10) 32
```

`let` is a syntactic sugar

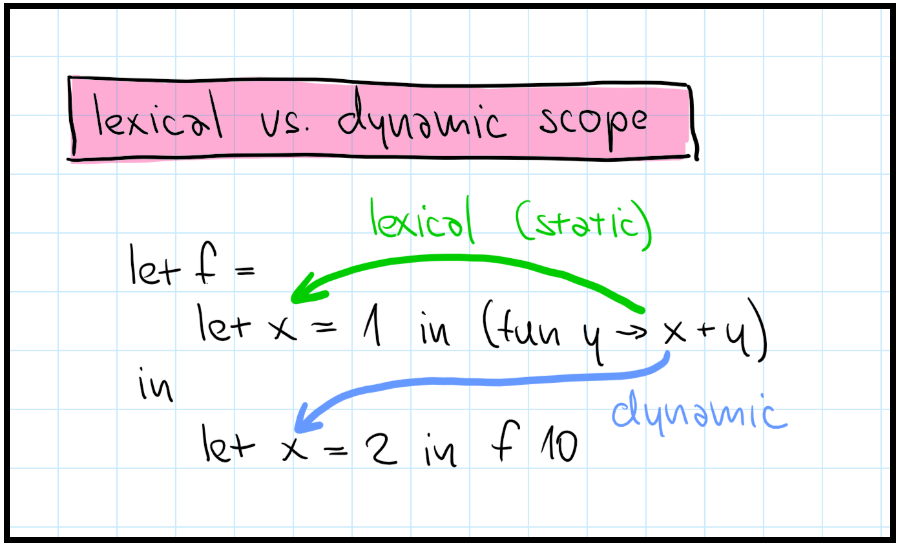
Everything (`if` and `let` too) is an expression

Functions that return functions (currying) if you need multiple parameters

Charles University

TinyML

A bit of theory



lexical vs. dynamic scope

```
let f =
    let x = 1 in (fun y -> x + y)
in
    let x = 2 in f 10
```

Variable scoping

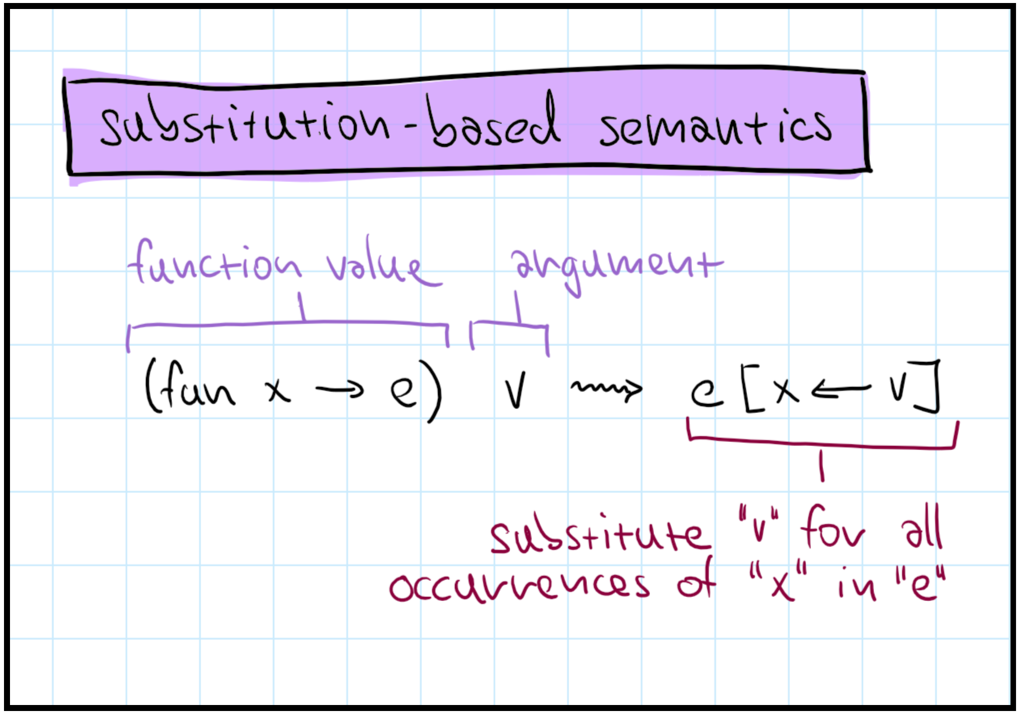
Lexical

Based on static block structure in code

Function value needs to capture variables (closure)

Dynamic

Based on dynamic evaluation structure



substitution-based semantics

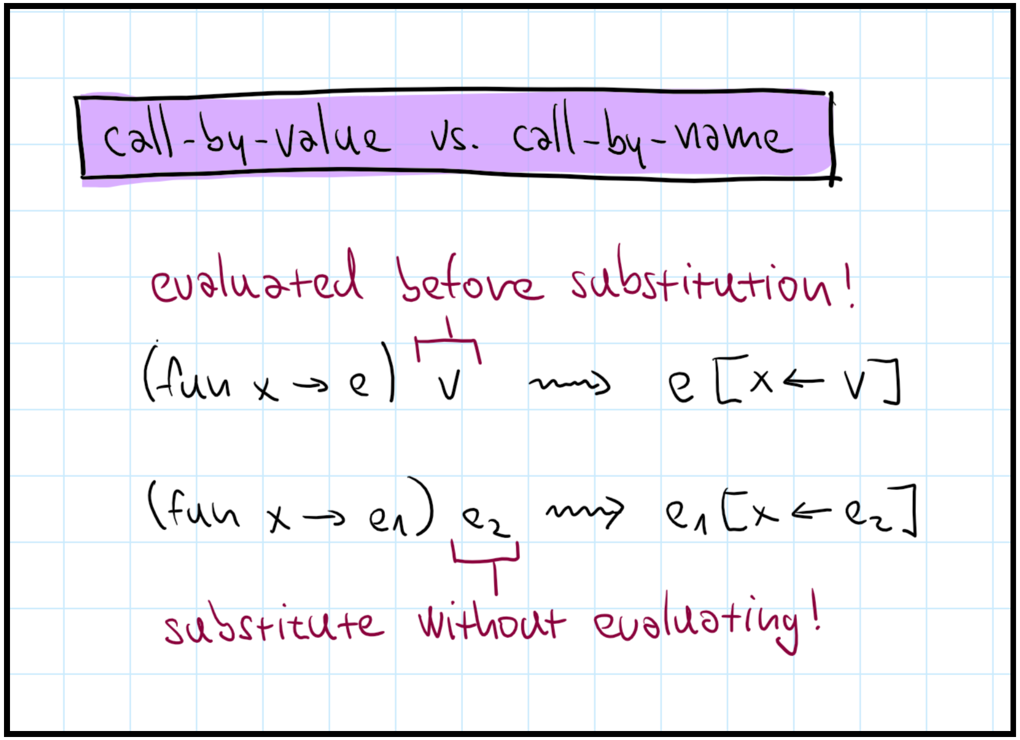
$$\overbrace{(\text{fun } x \to e)}^{\text{function value}}\ \overbrace{v}^{\text{argument}} \rightsquigarrow \underbrace{e[x \leftarrow v]}_{\text{substitute "v" for all occurrences of "x" in "e"}}$$

Operational semantics

Formally specify how expression evaluate

Substitution-based

We do not need variable context!



call-by-value vs. call-by-name

evaluated before substitution!

$$(\text{fun } x \to e)\ \overbrace{v} \rightsquigarrow e[x \leftarrow v]$$

$$(\text{fun } x \to e_1)\ \underbrace{e_2} \rightsquigarrow e_1[x \leftarrow e_2]$$

substitute without evaluating!

Call-by-name vs. call-by-value

Call-by-value (strict)

Evaluates function arguments first (ML)

Call-by-name (lazy)

Evaluates arguments when needed (Haskell)

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

Closing

A tiny functional language interpreter

Charles University

Conclusions

A tiny functional programming language interpreter

- Distinguishing `Value` and `Expression`
- Recursive function with variable scope
- Call-by-value and lexical variable scoping!
- Nice constructor and destructor symmetry

Tomáš Petříček, 309 (3rd floor)

✉ petricek@d3s.mff.cuni.cz

➜ https://tomasp.net | @tomaspetricek

➜ https://d3s.mff.cuni.cz/teaching/nprg077

Charles University