Functional Programming with OCaml

CSC 310 - Programming Languages

Functional Programming

- Functional programming
  - defines computations as mathematical functions
  - discourages use of mutable state
- State: the information maintained by a computation

Functional versus Imperative

- Function languages
  - Higher level of abstraction
  - Immutable state
  - Easier to develop robust software
- Imperative languages
  - Lower level of abstraction
  - Mutable state
  - More difficult to develop robust software

Imperative Programming

- Commands specify how to compute by destructively changing state
- The fantasy about changing state (mutability)
  - It is easy to reason about; the machine does this, then this, ...
- The reality
  - Machines are good at complicated manipulation of state
  - Humans are not good at understanding it

Imperative Programming

- Mutation breaks referential transparency, the ability to replace an expression with its value without affecting the result
- Problem: there is no single state
  - Programs have many threads, spread across many cores, spread across many processors, spread across many computers...
  - each with its own view of memory
- Difficult to examine a piece of code and reason about its behavior

Functional programming

- Expressions specify what to compute
  - Variables never change value (like mathematical variables)
  - Functions (almost) never have side effects
- The reality of immutability
  - No need to think about local state
  - Can perform local reasoning and assume referential transparency

ML-style (Functional) Languages

- ML (Meta Language)
  - University of Edinburgh, 1973
- Standard ML
  - Bell Labs and Princeton, 1990
- OCaml (Objective CAML)
  - INRIA, 1996
- Haskell (1998)
  - lazy functional programming

Key Features of ML

- First-class functions
  - Functions can be parameters to other functions, return values from functions, and stored as data
- Favor immutability
- Data types and pattern matching
- Type inference
  - No need to write types in the source program
  - Supports parametric polymorphism
- Exceptions
- Garbage collection

Why Study Functional Programming

Function languages predict the future:

- Garbage collection
  - LISP (1958), Java (1995), Python 2 (2000)
- Parametric Polymorphism (generics)
  - ML (1973), SML (1990), Java 5 (2004), Rust (2010)
- Higher-order functions
  - LISP (1958),
- Type Inference
  - ML (1973), C++11 (2011), Java (2011), Rust (2010)
- Pattern matching
  - SML (1990), Scala (2002), Rust (2010)

OCaml Compiler

- One OCaml compiler is `ocamlc`
  - Produces `.cmo` (compiled object) and `.cmi` (compiled interface) files
  - By default, links and produces an executable named `a.out`
- Another OCaml compiler is `ocamlopt`
  - Produces `.cmx` files, which contain native code
  - Faster executables, but not platform independent

OCaml Compiler: Multiple Files

- Suppose we have a `main.ml` file that depends on functions defined in a `util.ml` file
- Compile both together

```
ocamlc util.ml main.ml
```

- Compile both separately

```
ocamlc -c util.ml
ocamlc util.cmo main.ml
```

OCaml Top-level

- The top-level is a read-eval-print loop (REPL) for OCaml
- The `ocaml` program starts the top-level

```
$ ocaml
OCaml version 4.14.0
Enter #help;; for help.

# print_string "Hello world!\n";;
Hello world!
- : unit = ()
```

- To exit the top-level, type Control-D

Loading Code Files into the Top-level

- Load a file into the top-level

  `#use "filename.ml"`

- `#use` processes the file a line at a time

OPAM: OCaml Package Manager

- `opam` is the OCaml package manager
  - Manages libraries and different compiler installations
- Common packages to install
  - `ounit`, a unit testing framework
  - `utop`, a top-level with extra features
  - `dune`, a build system for larger projects

Building Projects with `dune`

- `dune` automatically finds dependencies and invokes the compiler and linker
- A `dune` file is similar to a `Makefile`
- `dune` can run a project's test suite with "`dune runtest`"

A Note on ;;

- ;; ends an expression in the top-level
  - basically says "evaluate the expression"
  - not used in the body of a function
  - not needed after each function definition
- ;; should not be used in program source code
- There is also a single semicolon in OCaml
  - Useful when programming imperatively with side effects

Recall: Syntax and Semantics

- The *syntax* of a programming language refers to structure of the language, that is, what constitutes a legal program.
- The *semantics* of a programming language refers to the meaning of a legal program.
- Additionally, *idioms* are conventional ways to use a language well

Expressions

- Expressions are the basic building block
- Every expression has
  - Syntax
    - Here the metavariable e to denotes an arbitrary expression
  - Semantics
    - Type checking rules (static semantics)
    - Evaluation rules (dynamic semantics)

Values

- A value is an expression cannot be evaluated any further
  - Here the metavariable v denotes an arbitrary value
- Evaluating an expression means computing it until it is a value.
- Examples
  - 42 is a value
  - $42 + 1$ is an expression that evaluates to 43

Types

- Types classify expressions
  - The set of values an expression could evaluate to
  - Here the metavariable t denotes an arbitrary type
- The expression e has type t if e will (always) evaluate to a value of type t
- Write `e : t` to denote e has type t
  - Determining that e has type t is called type checking

If Expressions

- Syntax:

```
(if e1 then e2 else e3) : t
    |       |       |
    v       v       v
  :bool     :t      :t
```

- Type checking: the if expression type checks if `e1` has type `bool` and both `e2` and `e3` have the type t

If Expressions: Type Checking and Evaluation

```
# if 1 > 2 then "a" else "b";;
- : string = "b"
# if true then 1 else 2;;
- : int = 1
# if false then 3 else "hello";;
Error: This expression has type string but an expression wa
         int
```

Function Definitions

- OCaml functions are like mathematical functions; compute a result from provided arguments
- Example:

```
let rec fact n =  (* rec is needed for recursion *)
  if n = 0 then   (* = is structural equality *)
    1
  else
    n * fact (n-1)
```

Type Inference

- A variable declaration does not need to be annotated with a type; the type can be inferred
- Type inference happens as part of type checking
  - Determines a type satisfies all constraints
- In the previous example, `n` has type `int` because the = operator takes two `int` expressions and returns a `bool` expression, so `n` must be an `int` for `n = 0` to type check

Function Application

- Syntax: `f e1 ... e1`
  - Parentheses are not required around arguments
  - No commas; use spaces instead
- Evaluation
  - Find the definition of `f`
  - Evaluate the arguments `e1 ... en` to values `v1 ... vn`
  - Substitute arguments `v1 ... vn` for params `x1 ... xn` in the body, `e`
    - call the resulting expression `e'`
  - Evaluate `e'` to value `v`, which is the final result

Example: Function Evaluation

- `fact 2`
- `if 2 = 0 then 1 else 2 * fact (2-1)`
- `2 * fact 1`
- `2 * (if 1 = 0 then 1 else 1 * fact(1-1))`
- `2 * 1 * fact 0`
- `2 * 1 * (if 0 = 0 then 1 else 0 * fact(0-1))`
- `2 * 1 * 1`
- `2`

Function Types

- In OCaml, `->` is the function type constructor
  - Type `t1 -> t` is a function with argument or *domain* type `t1` and return or *range* type `t`
  - Type `t1 -> t2 -> t` is a function that takes two inputs, of types `t1` and `t2`, and returns a value of type `t` (kind of – more on this later)
- Example:

```
# not;;
- : bool -> bool = <fun>
# (+);;
- : int -> int -> int = <fun>
```

Type Checking Function Application

- Syntax: `f e1 ... en`
- Type checking
  - If `f: t1 -> ... -> tn -> u`
  - and `e1:t1, ... en:tn`
  - then `f e1 ... en : u`

Type Checking Function Definition

- Syntax: `let rec f x1 ... xn = e`
- Type checking
  - Conclude the `f : t1 -> ... -> tn -> u` if `e:u` under the assumptions:
    - `x1:t1`, ... `xn:tn` (arguments with their types)
    - `f : t1 -> ... -> tn -> u` (for recursion)

Type Annotations

- The syntax `(e : t)` asserts that "`e` has type `t`"
  - This can be added (almost) anywhere
- Examples:

```
let (x : int) = 3

let fn (x : int) : float =
  (float_of_int x) *. 2.14
```

- Checked by the compiler – very useful for debugging