OCaml Trader

Patrick Flanagan
Jane Street

(HT Yaron Minsky, Marcin Sawicki)
Agenda

❖ Functional programming and OCaml
❖ Jane Street and people (including you!)
❖ Motivating examples
Functional Programming

Traditionally (John Hughes):

- no side effects (purity)
- higher-order functions and functors
- laziness
Classical Applications

- compilers
- AI (Lisp)
- formal validation of code
- automatic theorem proving
Syntax Tree

module Variable : sig type t end = struct
  type t = string
end

module Expression = struct
  type t =
    | Const of int
    | Var of Variable.t
    | Neg of t
    | Sum of t * t
    | Product of t * t
end

let five_plus_six = Sum ((Const 5), (Const 6))

(* 5 + 6 *)
module **Bool_expression** = struct

  type t =
  | Less_or_equal of Expression.t * Expression.t
  | Not of t
  | And of t * t
  | Or of t * t

end

let between_four_and_six =
  And (Less_or_equal (Const 4, Var "foo"), Less_or_equal (Var "foo", Const 6))

(* 4 <= foo && foo <= 6 *)
module Instruction = struct
    type t =
        | Assign of Variable.t * Expression.t
        | Print of Expression.t
        | While of Bool_expression.t * t
        | If_then_else of Bool_expression.t * t * t
        | Block of t list
end
let prog =
  Block [
    Assign ("foo", Const 5);
    While (Less_or_equal (Const 1, Var "foo"),
      (Block [
        Print (Var "foo");
        Assign (Var "foo", (Sum (Var "foo", Neg (Const 1))));
      ]))
  ];;

(* { foo = 5;
    while (1 <= foo); { 
      print foo;
      foo = foo + (-1);
    }
  } *)
Algebraic Datatypes

- available in languages like OCaml, SML, and Haskell
- products (tuples and records) are like C records
- variants are like C unions
- but they compose better
Who am I?
What does Jane Street do?

- Proprietary quantitative trading firm
- Trading (buying and selling) financial securities
- Focusing on technology, using OCaml
- Making markets ("market making", both buying and selling)
- Engaging in arbitrage
Market Participants

- investor
- speculator
- market maker
- arbitrageur
Market Participants

- investor
- speculator
- market maker
- arbitrageur
Our Needs

- correctness
- speed (but not for speed’s sake)
- correctness!!!
- agility of code writing and modification
- code must be easy to read (correctness!!!!)
Functional Programming

Traditionally (John Hughes):

- no side effects (purity)
- higher-order functions and functors
- laziness
Functional Programming

Our take (Yaron Minsky):

❖ expressive static types (with inference)
❖ higher-order functions and functors
❖ no side effects (purity)
❖ laziness
Laziness

- Peano numbers
- terminate evaluation early
- optimize compilation of programs, but…
- unpredictable (non-intuitive) evaluation
Purity

- all context / environment readily apparent (readability)
- object-oriented programming
Higher Order Functions

- compose control structures (compose code vs. data)
- avoid code duplication (fewer bugs)
- increase complexity without decreasing readability
fold

// sum the elements in a list

int sum(int array list) {
    sum = 0;
    for i in list; do
        sum = sum + i;
    done;
    return sum;
}
fold

list = [1; 2; 3; 4];
printf "%d
%!" (sum(list));
// "10"
// multiply the elements in a list

int product(int array list) {
    product = 1;
    for i in list; do
        product = product * i;
    done;
    return product;
}
fold

list = [1; 2; 3; 4];
printf "%d\n%!" (product(list));
// "24"
// sum a list
int sum(int array list) {
    sum = 0;
    for i in list; do
        sum = sum + i;
    done;
    return sum;
}

// multiply a list
int product(int array list) {
    product = 1;
    for i in list; do
        product = product * i;
    done;
    return product;
}
// fold over a list

int fold(int array list, int init, fun operate) {
    accumulator = init;
    for i in list; do
        accumulator = operate(accumulator, i);
    done;
    return accumulator;
}
fold

// fold over a list
list = [1;2;3;4]
sum(list) = fold(list, 0, (+)) // = 10
product(list) = fold(list, 1, (*)) // = 24
concat(list)
  = fold(list, "", (fun (s,i) ->
    s ^ int_to_string i))
  // = "1234"
Expressive Static Types

- real life (not just in finance) is complex and full of special cases
- useful code models the real world well
- variant types are a helpful tool to achieve this
‘a option

let div ~numerator ~denominator =

(* throws DivisionByZeroExn *)

numerator / denominator
type 'a option =
    | Some of 'a
    | None
let safe_div ~numerator ~denominator =
    if denominator <> 0 then
        Some (numerator / denominator)
    else
        None
val safe_div : numerator:int -> denominator:int -> int option
let print_div ~numerator ~denominator =

  match safe_div ~numerator ~denominator with
  | Some x -> Printf.printf "result = %d\n" x
  | None   -> Printf.printf "error: division by 0\n"
trading

type dir = Buy | Sell

let sign = function
    | Buy -> 1
    | Sell -> -1

type t =
    | Ack
    | Out
    | Fill of int * dir
let update_position t position =
  let delta =
    match t with
    | Ack -> 0
    | Out -> 0
    | Fill (size, dir) -> sign dir * size
  in
  position + delta
type dir = Buy | Sell

let sign = function
  | Buy -> 1
  | Sell -> -1

type t =
  | Ack
  | Out
  | Fill of int * dir
type dir = Buy | Sell

let sign = function
  | Buy -> 1
  | Sell -> -1

type t =
  | Ack
  | Out
  | Fill of int * dir
  | Bust of int * dir
let update_position t position =
  let delta =
    match t with
    | Ack -> 0
    | Out -> 0
    | Fill (size, dir) -> sign dir * size
  in
  position + delta
let update_position t position =
let delta =
  match t with
  | Ack
  | Out -> 0
  | Fill (size, dir) -> sign dir * size
  (* compile error--a missing case: Warning 8: this pattern-matching is not exhaustive. Here is an example of a value that is not matched: Bust (_, _)

File "kod.ml", line 148, characters 6-21: *)
in position + delta
let update_position t position =
    let delta =
        match t with
        | Ack -> 0
        | Out -> 0
        | Fill (size, dir) -> sign dir * size
    in
    position + delta
let update_position t position =
  let delta =
    match t with
    | Ack
    | Out -> 0
    | Fill (size, dir) -> sign dir * size
    | Bust (size, dir) -> sign dir * -size
  in
  position + delta
network connection status (bad)

type state =
  | Connecting
  | Connected
  | Disconnected

type t = {
  state: state;
  server: Inet_addr.t;
  last_ping_time: Time.t option;
  last_ping_id: int option;
  session_id: string option;
  when_initiated: Time.t option;
  when_disconnected: Time.t option;
}
type connecting = {
  when_initiated: Time.t;
}

type connected = {
  last_ping: (Time.t * int) option;
  session_id: string;
}

type disconnected = {
  when_disconnected: Time.t;
}

type state =
| Connecting   of connecting
| Connected    of connected
| Disconnected of disconnected

type t = {
  state: state;
  server: Inet_addr.t;
}
public static int binarySearch(int[] a, int term)

Returns:

index of the search term, if it is contained in the array; otherwise, (-(insertion point) - 1). The insertion point is defined as the point at which the term would be inserted into the array: the index of the first element greater than the term, or a.length if all elements in the array are less than the specified term. Note that this guarantees that the return value will be >= 0 if and only if the term is found.
return value (OCaml)

val binary_search:

    'a array
    -> term:'a
    -> [ `Found_at of int
        | `Not_found__insertion_point_at of int ]
return value (OCaml)

assert (  
    match binary_search a ~term with  
    | `Found_at idx -> a.(idx) = term  
    | `Not_found__insertion_point_at idx ->
      (idx = 0          || a.(idx - 1) < term)  
      && (idx = Array.length a || a.(idx) > term)
other interesting topics

- Async
- Incremental / Paralink
- Zero
- Iron
Further Reading

- **much code**
  - https://janestreet.github.io/

- **“core” library**
  - https://github.com/janestreet/core

- **async**

- **incremental**
  - https://blogs.janestreet.com/introducing-incremental/
We’re hiring!

janestreet.com/apply