CSC 330 / Spring 2023

OCaml
Patterns, Tail Recursion, Exceptions

Ibrahim Numanagić /nu-ma-nagg-ich/

based on Dan Grossman's lecture materials

Nested patterns

Nested patterns

- We can nest patterns inside other patterns
  - Just like we can nest expressions as deep as we want
  - Anywhere a variable can appear in our current patterns, we can put a pattern
  - Often avoids hard-to-read nested match expressions
- So full meaning of pattern-matching is to compare a pattern against a value for the *same shape* and bind variables to the *right parts*
  - More precise recursive definition coming after some examples

Code demonstration

Nested patterns

- Nested patterns are used in different contexts to express algorithms concisely and elegantly
  - Match on multiple things with same shape at same time
  - Go multiple levels into data structure at once
  - Use wildcard patterns when you do not need the data
  - Make match expressions that look like tables

Nested match expressions?

- Sometimes a match expression inside a match expression is a missed opportunity for nested pattern matching

```
match xs with
| [] -> 0
| x :: xs' -> match xs' with ...
```

- Other times it is necessary because you need to first compute with data extracted via the outer match

```
match xs with
| [] -> 0
| x :: xs' -> match f x s' with ...
```

- ⚠️: sometimes you need to wrap inner `match` statements with parentheses; otherwise you can end up with spectacularly confusing errors!

Patterns: generalization

- **Syntax:**

```
match e with p1 -> e1 | ... | pN -> eN
let p1 = e in e1
let [rec] f p1 = e1
```

  where `e, e1, ..., eN` are expressions, and `p1, ..., pN` are patterns

- **Pattern syntax:** A pattern is one of:
  - `x`: a variable
  - `_`: a wildcard
  - `(p1, ..., pN)`: a tuple of patterns
  - `C p`: a constructor `C` applied to a pattern `p`

> - Patterns are **NOT** expressions (though they kinda look like expressions)!

Match expressions: evaluation

Will now focus on match expressions, others are similar

- We need two things:
  1. Semantics of matching a pattern `p` to a value `v`
     - Does it, or does it not match?
     - If it matches, creates a set of (local) bindings
  2. Use (1) to define match expressions' semantics

Pattern-matching revealed

Yet another elegant recursive definition:

- Given pattern `p` and value `v`:
  - if `p` is a variable `x`, the match succeeds and `x` is bound to `v`;
  - if `p` is `_`, the match succeeds and no bindings are introduced;
  - if `p` is `(p1, ..., pN)` and `v` is `(v1, ..., vN)`, the match succeeds if and only if `p1` matches `v1` , and …, and `pN` matches `vN` (the bindings are the union of all bindings from the submatches);
  - if `p` is `C p1`, the match succeeds if and only if `v` is `C v1` (i.e., the same constructor) and `p1` matches `v1` (the bindings are the bindings from the submatch)

Examples

- `a :: b :: c :: d` matches all lists with $\geq$ 3 elements
- `a :: b :: c :: []` matches all lists with exactly 3 elements
- `((a, b), (c, d)) :: e` matches all non-empty lists of pairs of pairs

Match expressions: semantics

- **Evaluation:**

```
match e with p1 -> e1 | ... | pN -> eN
```

1. Evaluate `e` to `v`
2. Check in `p1`, `p2`, … order; find first `pi` that matches
3. Evaluate `ei` in environment extended with bindings from the match
4. Result of (3) is the overall result

Note: This is the *semantics*, but the implementation can *optimize* with concepts like binary search

Match expressions: type checking

- **Type checking:**

```
match e with p1 -> e1 | ... | pN -> eN
```

  - Type-check `e` to some type `t`
  - Each `pi` must match against some value of type `t`
  - Similar pattern, type matching definition to get the types of bindings
    - Typecheck `ei` in a static environment that has those bindings
  - All `e1, ..., eN` must have the same type `t2`, which is the overall type
  - Also, some fantastic features:
    - Error if a `pi` can never match because of earlier patterns (dead code)
    - Warning if a value of type `t` could match no `pi` (incomplete match)
      - (With nested patterns, these require some fairly fancy algorithms)

Code demonstration

Tail Recursion

Recursion

- Now have lots of practice with recursion
  - *Code follows the data*: recursive data processed recursively!
- No need for loops and assignment statements. Recursion often easier:
  - e.g., processing trees, appending lists
- But every recursive call needs to get "fresh space" for arguments and local variables, separate from the caller's "space"
- **Next**: let's refine our mental model of how calls execute (call stacks)
  - And see how recursion can be *as efficient as a loop*!

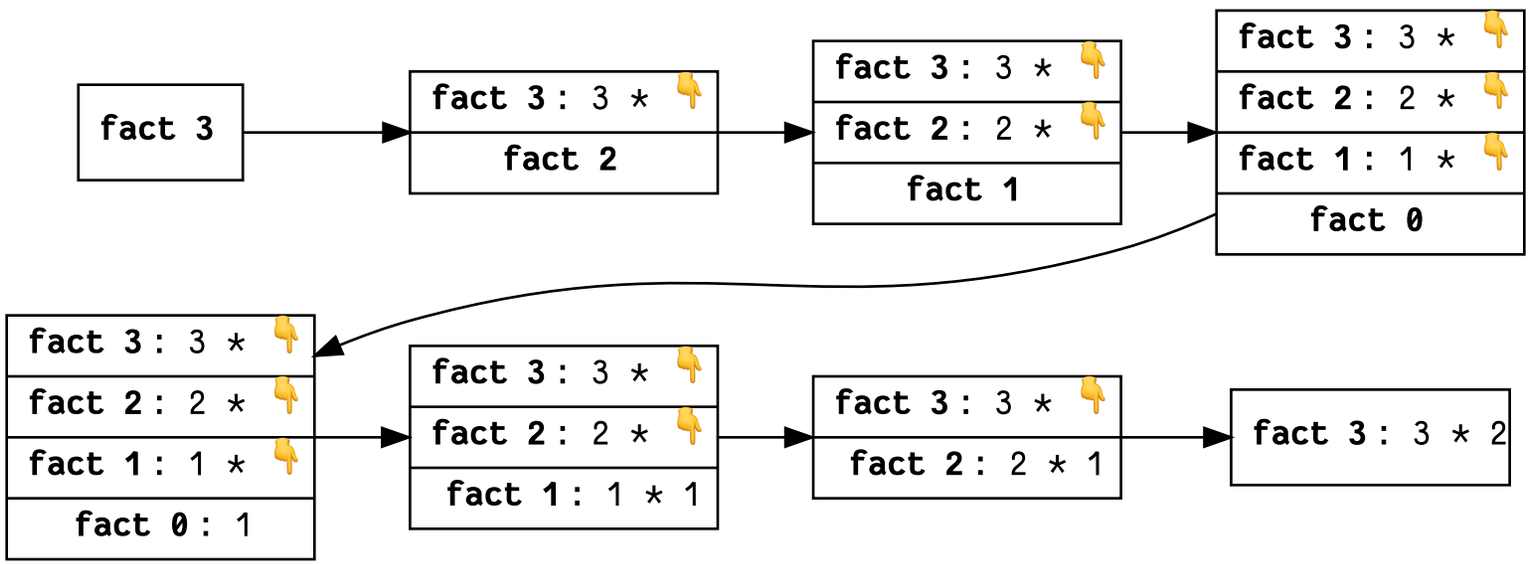
Call stacks

- While a program runs, there is a **call stack** for all function calls that have started but not yet completed
  - Calling `f` pushes a stack frame for `f` on the stack
  - When a call finishes, its stack frame is popped from the stack
  - Recursion: Can have more than one stack frames be for calls to the same function
- A stack frame stores information: values of parameters, values of local variables, and "what's left to do" for the function body

> - Function calls finish in the reverse order that they start!

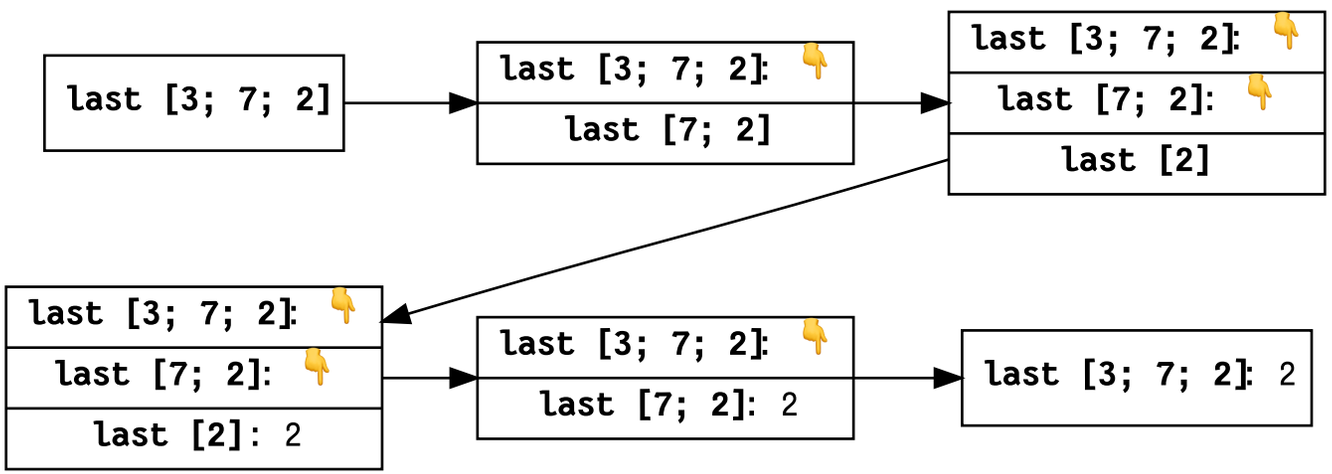
Example

```
let rec fact n =
  if n = 0 then 1 else n * fact(n - 1)
let x = fact 3
```



Example

```
let rec last xs =
  match xs with
  | x :: [] -> x
  | _ :: xs' -> last xs'
let x = last [3; 7; 2]
```

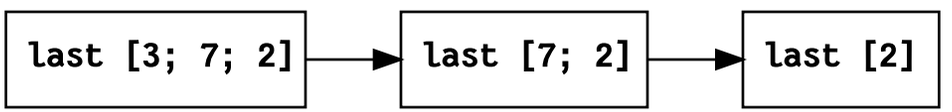


Do we *always* need a stack frame?

- **No!** It is unnecessary to keep around a stack-frame just so it can get a result from a function call and immediately return it "as is"
  - A call that will be the caller's answer *as is* is called a **tail call**
  - A recursive function where all recursive calls are tail calls is called **tail recursive**
- OCaml handles tail calls specially: **tail-call optimization**
  - Pops caller's frame **before** the call
  - This can take $O(n)$ space usage down to $O(1)$ 🤯
  - No *stack overflow* issues anymore!
  - Along with other optimizations, as efficient as a loop

`last` is tail-recursive!

```
let rec last xs =
  match xs with
  | x :: [] -> x
  | _ :: xs' -> last xs'
let x = last [3; 7; 2]
```



Making functions tail-recursive

- *Sometimes* we can rewrite our algorithm to be tail-recursive:

```
let fact n =
  let rec loop (n, acc) =
    if n = 0 then acc else loop (n - 1, acc * n)
  in
  loop(n, 1)
let x = fact 3
```

- *Sometimes* we can't easily or profitably do so:
  - processing a tree (two recursive calls can't both be tail calls)
  - if order matters (e.g., concatenating a string list)

Methodology

- There is a *methodology* to guide the transformation to a tail-recursive function:
  - Create a helper function that takes an **accumulator**
  - Old base case becomes initial accumulator
  - New base case becomes final accumulator
- Up to you to reason about whether this works
  - Example: for factorial, it is crucial that *multiplication is associative*

Code demonstration

Examples

- **Summing a list**: $O(1)$ stack space vs. length-of-list stack space
  - But notice both versions of factorial and list-sum take *time* linear in length of list
- **Reversing a list**: Tail-recursive version actually *much better*
  - Non-tail recursive version is *quadratic time* $O(n^2)$:
    - each recursive call uses `append`, which must traverse the list;
    - so total time for the appends is $1 + 2 + \cdots + (n + 1)$ which is roughly $\frac{n^2}{2}$
  - Moral: beware appending to lists, especially within outer recursion
  - Consing is constant time and fast (why?), so tail-recursive version is linear time

Moral: don't overdo it!

- Tail-recursion is not always feasible or elegant
- Tail-recursion is not always necessary: beware premature optimization and favor readability
- But where it is feasible and efficiency matters, it is a key tool for functional programmers to get the efficiency of loops
  - If you can write it as a loop, you can write it as a tail-recursive function: *updated values* are arguments to the tail call!
- Most functional languages, including OCaml, promise tail-call optimization

Precise definition of tail calls

- How do you know if a call is a tail call?
  - Humans: “Oh, I see the caller has no more work to do after it gets the result.”
  - Semanticists and compilers: a tail call is a function call that *is in tail position*
    - which is defined recursively over the syntax of expressions…

Tail position

- In `let f p = e`, the expression `e` is in tail position
- If `if e1 then e2 else e3` is in tail position:
  - then: `e2` and `e3` are in tail position;
    - but `e1` is **not** in tail position!
- If `let p = e1 in e2` is in tail position:
  - then: `e2` is in tail position
    - but `e1` is **not** in tail position
- If `e1 + e2`, the subexpressions are **not** in tail position
- … (more cases for every kind of expression)
- If `e` is **not** in tail position, then neither is any of its subexpressions

Exceptions

Exceptions

- Exception bindings declare new kinds of exception:

```
exception Whatever
exception BadNum of int
```

- `raise` expressions can throw an exception:

```
raise Whatever
raise (BadNum 1)
```

- `try ... with ...` expressions can catch exceptions:

```
try e with Whatever -> 0
try e with BadNum n -> n
```

Exceptions

- Exception bindings declare new kinds of exception:

```
exception Whatever
exception BadNum of int
```

- We *build* exceptions, which are just *values* (!) with these constructors:

```
Whatever
(BadNum 1)
```

- That is different than *raising* an exception with the `raise` expression:

```
raise Whatever
raise (BadNum 1)
```

A bit more precisely

- New exception bindings add new variants to "the one exception type" `exn`
- We build values of type `exn` just like we build variant types with constructors
  - Can pass them around, though that isn't that common
- We raise (throw) exceptions with `raise e`
  - **Type checking**:
    - `e` must have type `exn`
    - Result type is *any type you want (!)*
  - **Evaluation**:
    - Do not produce a result (!); but…
    - *cancel everything* and *pass the exception* produced by `e` to the nearest *try expression* on the call stack
      - and keep doing this while you can!

Try expressions

- `try ... with ...` expressions can *catch* exceptions:

```
try e1 with Whatever -> e2
try e1 with BadNum n -> e2
```

- **Evaluation**:
  - Just evaluate `e1` and that is the result
  - But if `e1` raises an exception *and* that exception *matches* the pattern, then evaluate `e2` and that is the result
- **Type checking**:
  - `e1` and `e2` must have the same type and that is the overall type

Code demonstration