CSCI 2041: OCaml Optimization Techniques

Chris Kauffman

Last Updated:
Mon Dec 10 09:04:27 CST 2018

Logistics

P5 Calculon

- Optional tests later today
- Due tomorrow

Lab 14

Review and Exit Survey

Goals

Examine optimizing using type annotations

Endgame

Date	Event
~~Wed 12/05~~	Lazy, Objects
	A5 Milestone
~~Fri 12/07~~	Object Systems
Mon 12/10	Optimization / Evals
Tue 12/11	Lab14: Review
	A5 Due
Wed 12/12	Last Lec: Review
Thu 12/13	Study Day
Mon 12/17	**Final Exam**
9:05am Sec 001	10:30am-12:30pm
1:25am Sec 010	1:30pm-3:30pm

Exercise: Consider the following Function

```
let cmp a b =
  a < b
;;
```

1. State the inferred type of the `cmp` function
2. Is it a polymorphic function? Why?
3. Speculate on whether there are any disadvantages to using this function

Answers: Consider the following Function

```
(* compare any two things, polymorphic '<' *)
let cmp_poly a b =
  a < b
;;
```

1. Type: `'a -> 'a -> bool`
2. Yes, it is polymorphic, any two types in
3. Being polymorphic it is very flexible, can be used with any type, but this requires **runtime type analysis**

The less-than operator must analyze values to determine what type they are to do comparison.

- This is not possible to do in normal OCaml
- Happens at the C level in the OCaml runtime system, the `compare_val` C function
- Cannot be optimized unless types are locked in early

Compare Comparisons

```
(* compare any two things, polymorphic '<' *)
let cmp_poly a b = a < b;;

(* compare only ints *)
let cmp_int (a:int) (b:int) = a < b;;

(* compare only strings *)
let cmp_str (a:string) (b:string) = a < b;;
```

- File `all_compare.ml` creates a main loop of random integer and string arrays
- Times runs of all pairwise comparisons using these three functions
- Examine source code for this file briefly

Exercise: Time Differences

```
> ocamlopt all_compare.ml

> ./a.out 5000
cmp_poly on ints
count: 12496306,  time: 0.3119 secs
cmp_int on ints
count: 12496306,  time: 0.1103 secs
cmp_poly on strings
count: 12496306,  time: 0.8095 secs
cmp_str on strings
count: 12496306,  time: 0.4314 secs
```

Speculate: why such a big difference in times?

Answers: Time Differences

- `cmp_poly` must perform an algorithm to determine types before beginning comparison

```
  if is_int(a)    then do_int_compare(a,b);
elif is_float(a)  then do_float_compare(a,b);
elif is_string(a) then do_string_compare(a,b);
etc.
```

- In contrast `cmp_int` and `cmp_string` know exactly which comparison instruction/function to use
- Opens up **inlining** opportunities for the compiler as well: call directly to the comparison functions
- Relevant to module functors as well: polymorphic comparison vs specific comparison functions

Example Functor Comparison

```
 1  type strpair = {
 2      first : string;
 3      second : string;
 4  };;
 5
 6  module PolyCmp = struct
 7    type t = strpair;;
 8    let compare = Pervasives.compare;;          (* polymorphic comparison *)
 9  end;;
10
11  module StringCmp = struct
12    type t = strpair;;
13    let compare a b =                           (* specific comparison *)
14      let diff = String.compare a.first b.first in
15      if diff=0 then
16        String.compare a.second b.second
17      else
18        diff
19    ;;
20  end;;
21
22  module PolySet    = Set.Make(PolyCmp);;
23  module StringSet  = Set.Make(StringCmp);;
```

`Pervasives.compare` vs Custom Comparison

- Using Module Functors like `Set.Make` must provide a comparison function
- Can always use `Pervasives.compare` but is usually more efficient to use a comparison function associated with a specific type

```
> ocamlopt set_test.ml

> a.out 200000
polyset search
count: 412,  time: 0.4179 secs
stringset search
count: 412,  time: 0.3144 secs
```

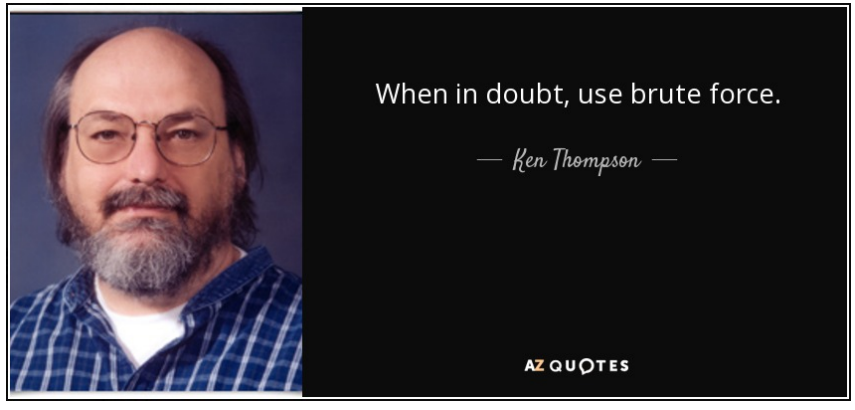
There are plenty of other opportunities to optimize bits and pieces of OCaml, but before you optimize, ask the question...

Caution: Should I Optimize?

- Optimizing program execution time usually costs human time
- Human time is valuable, don't waste it
- Determine if there is a NEED to optimize
- **Benchmark** your code - if it is fast enough, move on
- If not fast enough, use a **profiler** to determine where your efforts are best spent
- **Never sacrifice correctness** for speed

First make it work,
then make it right,
then make it fast.

- Kent Beck

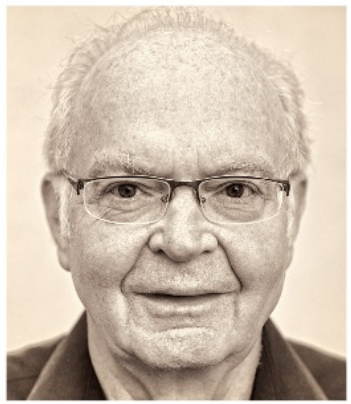


What to Optimize First

In order of impact

1. Algorithms and Data Structure Selection
2. Elimination of unneeded work/hidden costs
3. Memory Utilization
4. **Micro-optimizations**

"Premature optimization is the root of all evil" - Donald Knuth



Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: *premature optimization is the root of all evil.* **Yet we should not pass up our opportunities in that critical 3%.**
– Donald Knuth