Lecture 14: Runtime Safety, a Taste of OCaml

Steven Holtzen
s.holtzen@northeastern.edu

CS4400/5400 Fall 2024

1 Compiling Unsafely to MicroASM

- Now, let's compile calculator programs into MicroASM programs! How should we do that?
- This is similar in spirit to our Church encoding: we need to choose how we encode values of our calculator language into values of our assembly language.
- We hit a problem right away: our assembly language only has one kind of value, numbers. That makes it easy to encode numbers: those are naturally encoded to numbers. To encode Booleans, we map `#t` to 1 and `#f` to 0.
- Now that we have our encodings for values, we have to encode expressions. Let's start by encoding addition.
- Our approach to compiling calculator programs will be to *place the result of running each expression into a designated heap location*. The reason we store the result on the heap, and not in registers, is because we have a limited number of registers: we only have 3 of them, and some expressions might require more than 3 registers to be evaluated.
- For example, suppose we want to compile the calculator expression `(enum 10)`. We will compile this into an assembly program that places the number `10` at a designated place in the heap:

```
(list (Asetreg 1 10)
      (Astore 1 46))   ; store value 0 in location 46
```

- This program doesn't end with an `Ahalt` instruction, so it's not a complete program. In order to run it, we need to load the return value into register 0 and halt:

```
(list (Asetreg 1 10)
      (Astore 1 46)  ; store value 0 in location 46
      (Aload 0 46)   ; load value at location 46 into register 0
      (Ahalt))
```

- This suggests two functions we will need:
  - `calc->asm-h`, a helper function of type `calc -> (res asm addr)` that converts `calc` terms into a `result` type that contains (1) `asm`, a list of assembly instructions to run, and (2) `addr`, a number that contains the address of the result of running the program.
  - `calc->asm : calc -> listof asm` that calls `calc->asm-h` and extends the returned program by loading the result of running the final expression into register 0 and halting.

- Here is another example compilation. The program `(eadd (enum 10) (enum 20))` should compile to something like the following program (which may have different heap addresses):

```
(list (Asetreg 0 10)
      (Astore 0 46)  ; store value 10 in location 46
      (Asetreg 0 20)
      (Astore 0 47)  ; store value 20 in location 46
      (Aload 1 46)
      (Aload 2 47)
      (Aadd)
      (Astore 0 45)
      (Aload 0 45)
      (Ahalt))
```

- Now we are ready to implement these functions. The full source code is provided, but let's look at some small excerpts:

```
(define (unsafe-calc->asm-h c)
  (match c
    [(enum n)
     ; store n on a fresh heap location
     (define loc (fresh))
     (define prog (list (Asetreg 0 n)
                        (Astore 0 loc)))
     (res prog loc)]
    [(eadd e1 e2)
     ; compile e1 and e2
     (define loc (fresh))
     (match-define (res e1-prog e1-loc) (unsafe-calc->asm-h e1))
     (match-define (res e2-prog e2-loc) (unsafe-calc->asm-h e2))
     ; load e1-loc into reg1, e2-loc into reg2, add, then store the result into `loc`
     (define prog (list (Aload 1 e1-loc)
                        (Aload 2 e2-loc)
                        (Aadd)
                        (Astore 0 loc)))
     (res (append e1-prog e2-prog prog) loc)]
     ...))
```

```
;;; unsafe-calc->asm : calc -> asm
;;; compiles a calc program into an asm program
(define (unsafe-calc->asm c)
  (match-define (res prog loc) (unsafe-calc->asm-h c))
  ; make a program that loads loc into register 0
  (append prog (list (Aload 0 loc)
                     (Ahalt))))
```

- To implement `eand`, it is nearly identical to `eadd`, except we use multiplication instead of addition.
- Why is this called "unsafe" compilation? Intuitively, it's because the target language permits certain operations that are not allowed in the source language; this violates compiler correctness. Concretely, here is an example program that behaves differently in the source language and the target language:

  ```
  (eand (ebool #t) (enum 10)))
  ```

  This program causes a runtime error in the calculator interpreter, but evaluates to `10` using our compiler:

  ```
  > (interp-a (new-state (unsafe-calc->asm (eand (ebool #t) (enum 10)))))
  10
  ```

2 Compiling Safely to MicroASM

- We want to recover runtime safety for our compiler. How do we do this?
- Our solution will be to **tag** values that are stored on the heap with their type. This way we will be able to check whether or not a valid operation is being performed. This is how runtime safety is enforced in many widely-used programming languages, including Python, JavaScript, Racket, etc.
- We will need to raise a runtime error if we encounter an invalid tag. To do this, we will make use of the `Atrap v` instruction, which raises a runtime error if the contents of register `0` is not equal to `v`.
- Here is an excerpt of the safe version of our compiler that (1) adds tags to all values on the heap, and (2) checks for the correct tag whenever a value is retrieved from the heap:

```
(define int-tag 1234)
(define bool-tag 5678)

;;; helper for safely compiling calc to asm
;;; by convention, we will store (tag, value) on the heap any time we need to
(define (safe-calc->asm-h c)
  (match c
    [(enum n)
     ; store n on a fresh heap location, along with a num tag
     (define tag-loc (fresh))
     (define v-loc (fresh))
     (define prog (list (Asetreg 1 int-tag)
                        (Astore 1 tag-loc)   ; store an int-tag in tag-loc
                        (Asetreg 0 n)
                        (Astore 0 v-loc)))   ; store n in v-loc
     (res prog tag-loc)]
    [(eadd e1 e2)
     ; compile e1 and e2
     (define tag-loc (fresh))
     (define v-loc (fresh))
     (match-define (res e1-prog e1-tag-loc) (safe-calc->asm-h e1))
     (match-define (res e2-prog e2-tag-loc) (safe-calc->asm-h e2))
     (define e1-v-loc (+ 1 e1-tag-loc))
     (define e2-v-loc (+ 1 e2-tag-loc))
     (define prog (list
                   ; check that e1's tag and e2's tag are both the right type
                   (Aload 0 e1-tag-loc)
                   (Atrap int-tag)
                   (Aload 0 e2-tag-loc)
                   (Atrap int-tag)
                   ; now load the values, perform addition, and store the result
                   (Aload 1 e1-v-loc)
                   (Aload 2 e2-v-loc)
                   (Aadd)
                   (Astore 0 v-loc)
                   ; now store the tag
                   (Asetreg 0 int-tag)
                   (Astore 0 tag-loc)
                   ))
     (res (append e1-prog e2-prog prog) tag-loc)]
     ...))
```

3 Discussion: Pros and Cons of Runtime Safety

- Runtime safety complements type safety by preventing invalid behavior from occurring at runtime rather than statically.
- Runtime safety helps programmers localize bugs by failing early as soon as an illegal operation is performed.
- Programming in languages that enforce runtime safety is quite convenient since it does not require designing and conforming to a strict type system. For this reason, many widely-used programming languages today (like Python and JavaScript) rely on runtime safety over type safety.
- However, as we have seen, there is significant **overhead** associated with enforcing runtime safety:
  - *Memory overhead*: we need to store extra data (a tag) alongside every piece of data on the heap to ensure safety at runtime.
  - *Runtime overhead*: we need to perform extra instructions (comparing a tag, potentially trapping) every time a piece of data is accessed on the heap! This can be quite expensive in practice.

4 Practical Programming with Types: A Taste of OCaml

- At this point, it's very educational to see an example of an existing typed language that makes use of some of the typing features we've been exploring in this class so far, and how type systems can be translated into more practical languages.
- We will explore **OCaml**, a typed language that looks and behaves a lot like the simply-typed lambda calculus we have been studying. OCaml is not very widely used, but it has been very influential on many more widely-used languages: it has directly influenced the design of TypeScript, Rust, and Scala
- Let's quickly see how the typed language features we've explored so far – typechecking functions, pairs, and algebraic data types – work in OCaml.
- You can easily use OCaml by going to the following link: `https://try.ocamlpro.com/`
- On this webpage, you will see a split window similar to Racket with a *top-level* window and a *interaction window*. Let's type in the interaction window for now.
- Here is a small OCaml program that prints the number 5:

```
> 5
- : int = 5
```

- OCaml comments begin with `(*` and end with `*)`
- OCaml is an expression-oriented language, similar to Racket, but it has very different syntax: it uses an infix-style syntax. Here are some simple OCaml programs:

```
> 5 + 5
- : int = 10

> 5 = 10
- : bool = false

> "hello" = "world" ;;
- : bool = false
```

- OCaml has if expressions, Booleans, etc., which behave in an identical manner to the type systems we've explored so far for these features. Here is how they work syntactically:

```
> if true then 5 else 10
- : int = 5
```

- OCaml has a tyepechecker, so it will reject ill-formed programs:

```
> if 10 then 2 else 5 ;;
  Line 1, characters 3-5:
  Error: This expression has type int but an expression was expected of type bool
  because it is in the condition of an if-statement
```

- OCaml makes a similar compromise to our typechecker for `if`, and rejects if expressions whose branches return different types:

```
> if true then 10 else true ;;
  Line 1, characters 21-25:
  Error: This expression has type bool but an expression was expected of type int
```

- OCaml has support for lambdas and first-class functions. Functions are called in a manner similar to the surface syntax we've been using for the lambda calculus: by leaving a space between a function and its argument. Note that, in OCaml, parenthesis are used for disambiguating binding order, which is quite different from their meaning in Racket.

```
> (fun x -> x + 1) 10 ;;
  - : int = 11
```

  Notice that we were not required to annotate the type of the argument for this function! OCaml has **type inference**, which can infer from usage context that the argument must be of type integer here.

- If you want to annotate the type of a function's argument, you can do that with the following syntax:

```
> (fun (x:int) -> x + 1) 10 ;;
  - : int = 11
```

- OCaml supports let expressions as follows:

```
> let x = 10 in x + 5 ;;
- : int = 15
```

4.1 Pairs, Algebraic Data Types, and Pattern Matching

- OCaml supports pairs with the following syntax:

```
> (true, 10) ;;
- : bool * int = (true, 10)

> fst (true, 10)
- : bool = true

> snd (true, 10)
- : int = 10
```

- OCaml supports ADTs with the following syntax (which should be familiar to you!)

```
> type left_or_right = Inl of int | Inr of int ;;
type left_or_right = Inl of int | Inr of int

> Inl 10 ;;
- : left_or_right = Inl 10
```

  Note: OCaml requires that the first letter of each ADT is capitalized.

- OCaml supports pattern matching for breaking apart ADTs:

```
>  match (Inl 10) with
  | Inl v -> v + 10
  | Inr v -> v + 20 ;;
  - : int = 20
```

- OCaml's type checker will helpfully alert you if you are not handling each case in a pattern match:

```
> match (Inl 10) with
  | Inl v -> v + 101 ;;
- : int = 20
  1 Warning : this pattern-matching is not exhaustive.
  Here is an example of a case that is not matched:
  Inr _
```

5 An Interpreter in OCaml

- A nice way of studying a new programming language is to implement a program you're familiar with in it. Here is an implementation of calculator language:

```
  type expr =
  Num of int
  | Bool of bool
  | Add of expr * expr
  | And of expr * expr

type value =
  VBool of bool
  | VNum of int

let rec interp (e : expr) : value =
  match e with
  | Bool(b) -> VBool(b)
  | Num(i) -> VNum(i)
  | Add(e1, e2) ->
    (match (interp e1, interp e2) with
     | (VNum n1, VNum n2) -> VNum (n1 + n2)
     | _ -> failwith "runtime error")
  | And(e1, e2) ->
    (match (interp e1, interp e2) with
     | (VBool n1, VBool n2) -> VBool (n1 && n2)
     | _ -> failwith "runtime error")
```

- Then, we can try running some programs in the REPL:

```
 > interp (Add (Num 10, Num 20));;
- : value = VNum 30
```