Lecture 13: Runtime Safety

Steven Holtzen
s.holtzen@northeastern.edu

CS4400/5400 Fall 2024

1 Racket Contracts

- Today's lecture is about **runtime safety**: ensuring the safe behavior of programs *at runtime*
- What does "safety" mean here? In short, **runtime safety** means failing (i.e., raising an error) when an illegal operation is performed at runtime.
- Racket enforces runtime safety via its contract system. For instance, if we try to add a number and symbol, we get a contract error:

```
> (+ 10 'oops)
  +: contract violation
    expected: number?
    given: 'oops
```

- Racket has an extensive system for contracts that lets the programmer specify when certain behavior should or should not be legal at runtime.
- For example, so far in class we've been declaring our ASTs like this:

```
;;; type e =
(struct enum (n) #:transparent)
(struct eadd (e1 e2) #:transparent)
(struct ebool (v) #:transparent)
(struct eand (e1 e2) #:transparent)
```

Listing 1: The familiar calculator AST.

- One of the problems with this is that we are allowed to put any datatype we want inside of each of these structs and Racket won't raise a runtime error. For instance, I can write the following:

```
> (enum "oops")
(enum "oops")
```

- This is a problem because the bad behavior is not **localized**, which can make it hard to debug broken programs. If I put in invalid piece of data inside of a `enum`, I might not find out until much later on in the program that I went wrong. The principle of runtime safety says to fail *as soon as the illegal operation is performed*. Let's use Racket contracts to get this behavior.

- Racket has a special additional annotation `#:guard` that we can place inside a struct definition. After `#:guard`, you can provide a contract called a `struct-guard/c` (the convention `/c` denotes a contract in Racket). This contract expects a sequence of predicates as its arguments.[1] The principle is best illustrated by example:

```
(define (calc? e)
  (or (enum? e) (eadd? e) (ebool? e) (eand? e)))

;;; type e =
(struct enum (n) #:transparent #:guard (struct-guard/c number?))
(struct eadd (e1 e2) #:transparent #:guard (struct-guard/c calc? calc?))
(struct ebool (v) #:transparent #:guard (struct-guard/c boolean?))
(struct eand (e1 e2) #:transparent #:guard (struct-guard/c calc? calc?))
```

Listing 2: Calculator AST with contracts.

- What this contract ensures is that, whenever we create a `enum`, the `number?` predicate is called on the argument `n`; if `number?` returns `#f`, then a contract error is raised.
- We can define our own predicates to use in contracts. For instance, we can define a `calc?` predicate that checks whether or not its argument is a `calc` AST.
- We can also add contracts to our function definitions in the following way:

```
;;; add n1 to n2 and return the result
(define/contract (my-add n1 n2)
  (-> number? number? number?)
  (+ n1 n2))
```

- The above contract checks that first two arguments and return value of `my-add` satisfy the `number?` contract.
- We will use contracts like this from now on in order to enforce runtime safety of our Racket programs.

[1] A predicate is a function that returns a Boolean.

2 What is Runtime Safety?

- So far we've been discussing static typechecking where types are determined without running the program
- Some programming languages offer dynamic typechecking instead of or in addition to static typechecking, which checks that values are the correct type *at runtime*.
- An example is Python, which will determine at runtime whether or not you can perform an operation on two pieces of data:

```
>>> 1 + "two"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

- The process of checking at runtime that the correct type of values is used is called **dynamic safety** (or **runtime safety**).
- How is dynamic typechecking implemented in practice? Let's see!
- Let's consider again our tiny calculator language:

```
(define (calc? e)
  (or (enum? e) (eadd? e) (ebool? e) (eand? e)))

;;; type e =
(struct enum (n) #:transparent #:guard (struct-guard/c number?))
(struct eadd (e1 e2) #:transparent #:guard (struct-guard/c calc? calc?))
(struct ebool (v) #:transparent #:guard (struct-guard/c boolean?))
(struct eand (e1 e2) #:transparent #:guard (struct-guard/c calc? calc?))

(struct vbool (b) #:transparent #:guard (struct-guard/c boolean?))
(struct vnum (n) #:transparent #:guard (struct-guard/c number?))

(define (interp-e e)
  (match e
    [(enum n) (vnum n)]
    [(ebool b) (vbool b)]
    [(eand e1 e2)
     (vbool (vbool-b (interp-e e1))
            (vbool-b (interp-e e2)))]
    [(eadd e1 e2)
     (vnum (vnum-n (interp-e e1))
           (vnum-n (interp-e e2)))]))
```

- What are some examples of runtime safety errors? Well, one is running a program that tries to add a Boolean and a number:

```
> (interp (eadd (ebool #t) (enum 10)))
. error
```

- Racket's host semantics perform runtime safety for us, and prevents our program from continuing to run and going wrong in mysterious ways. So, we inherit runtime safety for our interpreter from our host semantics. But, *what if our host semantics don't enforce runtime safety*? How can we recover runtime safety in this setting? To explore this question, we will implement a small CPU and compile calculator programs to it.

3 MicroASM

- Let's implement a mini CPU language called MicroASM. MicroASM is a very simple register-based "abstract machine".
- The **CPU state** is the central semantic object for MicroASM: each MicroASM instruction updates the CPU state in some way. Here is the CPU state as a Racket struct:

```
;;; tracks the state of the CPU
;;; reg : a hashmap of number -> number
;;; heap: a hashmap of number -> number
;;; program: a list of program instructions
;;; insn: a boxed number, the index of the next instruction to be executed
(struct state (reg heap program insn) #:transparent
   #:guard (struct-guard/c hash? hash? (listof asm?) box?))
```

- Here is the syntax of MicroASM:

```
(define (valid-reg? k)
(and (integer? k)
     (>= k 0)
     (<= k 3)))

(define (valid-heap-loc? k)
  (and (integer? k)
       (>= k 0)
       (<= k 50000)))

(define (asm? a)
 (or (Aload? a) (Astore? a) (Asetreg? a) (Atrap? a) (Amul? a) (Aadd? a) (Ahalt? a)))

;;; type asm =
;;; load value at `addr` into `reg`
(struct Aload (reg addr) #:transparent #:guard (struct-guard/c valid-reg? valid-heap-loc
    ?))
;;; store register[reg] into heap[addr]
(struct Astore (reg addr) #:transparent #:guard (struct-guard/c valid-reg? valid-heap-
    loc?))
;;; set register[reg] = value
(struct Asetreg (reg value) #:transparent #:guard (struct-guard/c valid-reg? integer?))
;;; trap: gives a runtime error if register[0] != v, otherwise does nothing
(struct Atrap (v) #:transparent #:guard (struct-guard/c integer?))
;;; set register[0] = register[1] + register[2]
(struct Aadd () #:transparent)
;;; set register[0] = register[1] + register[2]
(struct Amul () #:transparent)
;;; terminate
(struct Ahalt () #:transparent)
```

Listing 3: Syntax for MicroASM

4 Semantics of MicroASM

- The MicroASM CPU works imperatively: it executes one instruction at a time, and mutates the CPU state after executing each instruction. The CPU will keep evaluating instructions until it hits a `Ahalt` instruction, at which point it will terminate and return the value contained in register 0.
- We will visualize running MicroASM programs using the **step arrow** $\rightarrow$, which simulates executing an instruction.
- Let's run a few programs using the step arrow to get a feel for how it works. First, let's run a simple program that (1) sets register 1 equal to 1, (2) sets register 2 equal to 2, (3) adds these two registers and returns the result:

```
(define prog (list (Asetreg 1 1)   ; set reg[1] = 1
                   (Asetreg 2 2)   ; set reg[2] = 2
                   (Aadd)          ; set reg[0] = reg[1] + reg[2]
                   (Ahalt)))       ; halt and return reg[0]
```

- This would result in the following **trace**, which simulates each step of evaluating the assembly program `prog`:

```
(state {} {} prog (box 0))
--> (state {1 ↦ 1} {} prog (box 1))
--> (state {1 ↦ 1, 2 ↦ 2} prog (box 2))
--> (state {0 ↦ 3, 1 ↦ 1, 2 ↦ 2} prog (box 2))
--> 3
```

- So far we've only used registers. However, we have a very limited number of those! Notice, our CPU only has 3 registers available. In order to manipulate more data, we need to store our data in places other than registers. This is where the heap comes in.
- This program was quite simple and doesn't involve the **heap**, which maps addresses to numbers. We can manipulate the heap by using the store and load instructions:

```
(define prog2 (list (Asetreg 1 1) ; set reg[1] = 1
                    (Astore 1 3)  ; store reg[1] into heap[3]
                    (Aload 0 3)   ; load heap[3] into reg[0]
                    (Ahalt)))
```

- Now we can step this program to show how the CPU state evolves with each instruction:

```
(state {} {} prog2 (box 0))
--> (state {1 ↦ 1} {} prog2 (box 1))
--> (state {1 ↦ 1} {3 ↦ 1} prog2 (box 2))
--> (state {0 ↦ 1, 1 ↦ 1} {3 ↦ 1} prog2 (box 3))
--> 1
```

- There is one more interesting instruction: `(Atrap v)`. This raises a runtime error if register 0 is not equal to `v`. We will use this later to implement runtime safety.

5 Implementing MicroASM

- The state makes use of a language feature we haven't seen before: *boxes*. Boxes are how we will deal with mutable state in Racket. A box contains a value that can be changed: `(box e)` creates a boxed value, `(unbox e)` gets the value inside the box, and `(set-box! e1 e2)` sets the boxed value in `e1` to be equal to `e2`. For example:

```
> (define my-counter (box 10))
> (unbox my-counter)
10
> (set-box! my-counter 20)
> (unbox my-counter)
20
```

- We will see more about boxing next lecture when we implement a typechecker for mutable state.

```
(define/contract (interp-a s)
  (-> state? number?)
  ; this is some handy syntax to combine `define` and pattern matching
  (match-define (state reg heap prog insn) s)
  (define cur-insn (list-ref prog (unbox insn)))
  (match cur-insn
    [(Aload r addr)
     (define v (hash-ref heap addr))
     (hash-set! reg r v)
     (set-box! insn (+ (unbox insn) 1))
     (interp-a s)]
    [(Astore r addr)
     (define v (hash-ref reg r))
     (hash-set! heap addr v)
     (set-box! insn (+ (unbox insn) 1))
     (interp-a s)]
    [(Asetreg r v)
     (hash-set! reg r v)
     (set-box! insn (+ (unbox insn) 1))
     (interp-a s)]
    [(Asetreg r v)
     (hash-set! reg r v)
     (set-box! insn (+ (unbox insn) 1))
     (interp-a s)]
    [(Aadd)
     (define v1 (hash-ref reg 1))
     (define v2 (hash-ref reg 2))
     (hash-set! reg 0 (+ v1 v2))
     (set-box! insn (+ (unbox insn) 1))
     (interp-a s)]
    [(Amul)
     (define v1 (hash-ref reg 1))
     (define v2 (hash-ref reg 2))
     (hash-set! reg 0 (* v1 v2))
     (set-box! insn (+ (unbox insn) 1))
     (interp-a s)]
    [(Atrap v)
     (define reg0-v (hash-ref reg 0))
     (set-box! insn (+ (unbox insn) 1))
     (if (equal? v reg0-v)
         (interp-a s) ; keep going
         (raise-trap-error) ; raise error
         )]
    [(Ahalt)
     ;;; return the contents of register 0
     (hash-ref reg 0)
     ]))
```

Figure 1: A MicroASM interpreter.