Heap-Allocated Data and Assignment Conversion

CIS531 — Fall 2025, Syracuse

Kristopher Micinski

Syracuse University
College of Engineering
& Computer Science

set!

- Racket feature, allows us to **mutate** any variable; of course, this is the norm in traditional imperative programming
- Also, (begin es … e-last) evaluates each in order and returns result of e-last
- Begin is only useful when you care about the side effects of es…
- In project 4 you will implement set!
  - The trick: use "boxing" to move variables to the heap, then every usage of the variable becomes a *dereference*; this allows variables to change over time

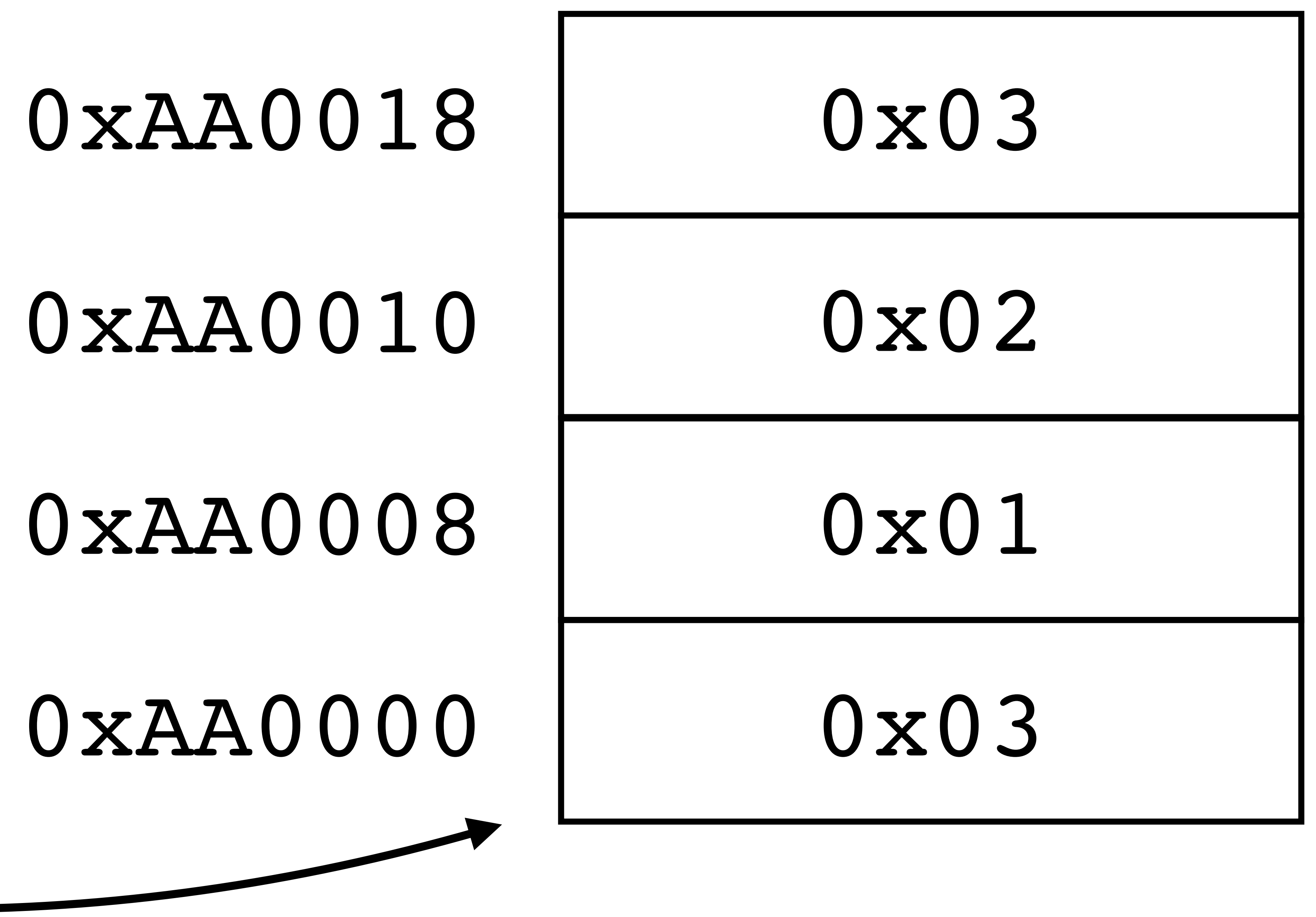
```
(let ([counter 0])
  (begin
    (displayln counter)  ; 0
    (set! counter (+ counter 1))
    (displayln counter)  ; 1
    (set! counter (* counter 2))
    (displayln counter)) ; 2
```

Vectors

- So far, we only have *one* type of data: a single 64-bit value
- In this project, we generalize this to *heap-oriented* data in the form of *vectors*
- A vector is simply a data array (64-bit chunks) prefixed by a length
- 64-bit length followed by a flat data array—simplest-possible thing!

```
typedef struct {
    int64_t len;
    int64_t data[];
} TinyVecPacked;
```

'#(1 2 3)



Represented in memory as…

Vector operations

- In Racket, several operations:
  - (make-vector e) — build a new (zeroed) vector of size e
  - (vector e0 …) — build a new vector of values e0, …
  - (vector-ref e0 e1) — dereference the vector e0 at index e1
  - (vector-set! e0 e1 e2) — mutably update the vector e0 at the index e1 w/ value e2

- In our case, we will support a thinner API:
  - (make-vector i) — has to be a constant
  - (vector-ref e0 i) — has to be a constant
  - (vector-set! e0 i e1) — has to be constant
- Can generalize this without much issue, but makes things a bit simpler for now…

Implementing make-vector

- Why not just allocate on stack?
  - Don’t want to allocate large data on stack, but more importantly…
  - We want to anticipate data that outlives its stack frame (next project…)
    - When the function returns, it pops its stack frame
    - If we return a reference to data on the stack, it will be **corrupted**
- Simplest *possible* thing—just use malloc
- Yes, this *will* leak memory—we will talk about garbage collection soon

```
TinyVecPacked* make_vector(int64_t len) {
  int64_t init_val = 0;
  TinyVecPacked* v = (TinyVecPacked*)malloc(sizeof(TinyVecPacked) + len * sizeof(int64_t));
  v->len = len;
  for (int64_t i = 0; i < len; i++) {
    v->data[i] = init_val;
  }
  return v;
}
```

Implementing vector-ref and vector-set!

- Can be implemented in a **very fast** but *unsafe* manner:
  - Vector ref is simply movq!
  - ```
    (let ([x (vector-ref e i)]) …)
    -> … move result into %rax …
       movq i(%rax), x
    ```
  - Vector-set! is also movq

- Issue: both of these are *unsafe*
    ```

Safe vector-ref and vector-set!

- If you read from / write to memory outside of the bounds of a buffer, bad things happen: memory corruption, segfault, security vulnerabilities, etc.
- If you <u>know</u> x *is a vector*, then you at least know that the size is at &x
- Together with a runtime exception handler (which kills the program), you can thus implement a **safe** vector-ref and vector-set!
  - First check valid access, then read/write as usual
  - *But* you need a branch to check if the read is in the bounds of the vector
  - (Extra code, stresses branch predictor, etc.)
- What if we don't even know if it's a vector? (Answer: can't say **anything**)

A motivation for type systems…

- This is why type systems matter: we need to know at least that something *is a vector* to even have a safe version of vectors
  - Otherwise we could treat an *integer* (or other primitive types) as a vector!

- A type system can also (sometimes) statically ensure that reads/writes are always to valid parts of the vector
  - Not always possible, because arithmetic is undecidable in general
  - Other program analyses may be used to accomplish this as well
  - May need run-time tagging on objects:
    - Top/lower bit says: "pointer or literal?" (OCaml: 63-bit integers…)

We have a few options to mitigate access to vectors safely…

- (A) Just be unsafe, assume that things have the right type (easy, fast, wrong)
  - If they don't, it's a segfault, user error, etc.
- (B) Type system to check shapes / sizes of vectors
  - <u>At least</u> need to know "vector or not vector," so we can emit safe checks
    - If we don't know this, **all** operations are destined to be unsafe
  - "Vector of a known size" allows us to emit optimized code (no branch)
- (C) Dynamically tag all runtime objects
  - Bottom 4 bits of an 8-byte-aligned value are ***always zero***
    - We can exploit this: four free bits!
  - Downside: always have to check tag before unpacking value
  - OCaml: unboxed 63-bit ints, very bottom bit answers: object or int?
    - Ints are very fast, but you give up one bit of representation…
    - Can always box ints, but then extra memory accesses

Assignment Conversion and Boxing

- *To implement set!*
  - Treat every **let** binding as an *allocation*:
    - `(let ([x e]) …) => (let ([x (box e)]) …)`
  - Treat every variable **reference** an an implicit dereference:
    - `x => (unbox x)`
- In our compiler, we'll implement box/unbox

Autoboxing / Unboxing in Java (Oracle Java SE docs)...

- Java automatically boxes integers and other primitives
- Automatic coercions of int => Integer
  - Wrapper class: “boxed” integer

Autoboxing and Unboxing

Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an `int` to an `Integer`, a `double` to a `Double`, and so on. If the conversion goes the other way, this is called *unboxing*.

Here is the simplest example of autoboxing:

```
Character ch = 'a';
```

The rest of the examples in this section use generics. If you are not yet familiar with the syntax of generics, see the Generics (Updated) lesson.

Consider the following code:

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 50; i += 2)
    li.add(i);
```

Although you add the `int` values as primitive types, rather than `Integer` objects, to `li`, the code compiles. Because `li` is a list of `Integer` objects, not a list of `int` values, you may wonder why the Java compiler does not issue a compile-time error. The compiler does not generate an error because it creates an `Integer` object from `i` and adds the object to `li`. Thus, the compiler converts the previous code to the following at runtime:

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 50; i += 2)
    li.add(Integer.valueOf(i));
```

Converting a primitive value (an `int`, for example) into an object of the corresponding wrapper class (`Integer`) is called autoboxing. The Java compiler applies autoboxing when a primitive value is:

- Passed as a parameter to a method that expects an object of the corresponding wrapper class.
- Assigned to a variable of the corresponding wrapper class.

Consider the following method:

```
public static int sumEven(List<Integer> li) {
    int sum = 0;
    for (Integer i: li)
```

(assignment-convert p)

- In P4, you will implement the pass (assignment-convert p)
- Basic idea: walk over any expression e:
  - Replace every variable x reference with a dereference: (vector-ref x 0)
  - Replace every let-binding (places where variables originate) with an allocation: (let ([x e]) …) becomes (let ([x (vector e)]) …)
    - We don’t have this vector form in our language
    - Instead, use a combo of vector followed by a vector-set!

(void) and Side-effecting expressions

- What should the return value of (vector-set!) be?
  - (let ([x (vector-set! y 2 z)]) …)
- Introduce a new value, (void), the "unit value."
- Expressions such as vector-set! and set! return (void)

- Another idea: since we know these expressions will be void, why bind them at all? Instead, just use…
  - (let ([_ (vector-set! 2 z)]) …)
  - This form of let says: execute RHS for the **effect**, but drop the return value
  - I use this form of let ***pervasively*** in project 4
    - ***CAUTION: Make sure to always match this first!!!***

(Begin ...)

- Notice that now we can have chains of expressions executed for their *effect*
- Instead, we can use begin; begin readily translates into nested (let ([_ ...]) ...)

```
(begin (set! tmp x)
       (set! y x)
       (set! x tmp)
        y)
```

```
(let ([_ (vector-set! tmp 0 x)])
  (let ([_ (vector-set! y 0 (vector-ref x 0))])
    (let ([_ (vector-set! x 0 (vector-ref tmp 0))])
      (vector-ref y 0))))
```

Allocation details

- Previously, we used **malloc** to allocate memory
  - https://danluu.com/malloc-tutorial/ (just something I found…)
  - Swiss-army-knife of memory allocation:
    - **But expensive!** ~100s of cycles, varies a *lot* depending on if we need to mmap

- Fact: whenever you read **any** memory address, you *pull in a whole cache line*
  - 64 bytes (8 register-sized words) on modern CPUs

- Fundamental tension of malloc: allocating varying-size objects is tough to do
  - Can often do much better if we know things about object's **shape** and **lifetime**
  - E.g., custom allocator for cons cells (pairs); However, modern allocators (like those in modern glibc) have fast-paths for small fixed-size objects
  - C++: custom allocators not uncommon for media-style applications

A Potential Criticism of This Approach

- One obvious criticism: all "local" variables will now be **allocated on the heap**
  - Is this "just the price we pay for set!?"
  - Answer: no, obviously not, in C local variables are **stack allocated**

- So why can't we simply stack allocate things as well?
  - Answer: we totally can do that in instances where we can show the data does not escape (get returned from the function)
  - In this case, there are no functions, so no risk of escaping
  - Can't return a buffer / vector—return value has to fit in register
  - So in general, we need to return references to things…
  - To make things **easy**, we **box everything** in P4
    - Possible to write an optimization pass which puts them on the stack
      - (Is this really faster than on the heap? They're both movqs…)
      - Answer: yes, better locality due to cache lines

What to know from this lecture

- make-vector, vector-ref, and vector-set! — mutable vectors
- How is make-vector implemented? (Malloc)
- What are the safety concerns around interacting w/ vectors wrt assembly?
  - Not even sure if it’s a **vector** without type checking
    - (Either runtime or static—trade offs)
  - Once we know it’s a vector, we can check length—avoid segfault
- What is “boxing” and how is it implemented?
- What does the “assignment-convert” pass do?
  - What happens to set! after this pass?