EECS 483: Compiler Construction

Lecture 12:
Heap Allocation

February 23
Winter Semester 2026

Reminder

Assignment 3 (Procedures) due on Friday

Learning objectives

Stack vs Heap Allocation

Mutable Arrays syntax and semantics

Implementation of Heap-allocated, large datatypes

Live Demo: Dynamic Typing Sandbox

State of the Snake Language

Adder: Straightline Code (arithmetic circuits)

Boa: local control flow (finite automata)

Cobra: procedures, extern (pushdown automata)

Snake v4: **Diamondback**

1. Add new datatypes, use dynamic typing to distinguish them at runtime
2. **Include heap-allocated variable-sized arrays, allowing for unrestricted memory usage**

Computational power: Turing complete

Extending the Snake Language

Diamondback: Arrays

```
def main(x):
  [x , x + 1, x + 2]
```

allocate an array with a statically known size

Extending the Snake Language

Diamondback: Arrays

```
def main(x):
  newArray(x)
```

allocate an array with dynamically determined size (elements initialized to 0)

Extending the Snake Language

Diamondback: Arrays

```
def main(x):
  let a = [x , -1 * x ] in
   a[0]
```

array indexing

Extending the Snake Language

Diamondback: Arrays

```
def main(x):
  let a = [x , -1 * x ] in
   let _ = a[1] := a[1] + 1 in
   a[1]
```

arrays can be mutably updated

Extending the Snake Language

Diamondback: Arrays

```
def main(x):
  let a = [x , -1 * x ] in
  length(a)
```

should be able to access the length of any array value

Extending the Snake Language

Diamondback: Arrays

```
def main(x):
  let a = [x , -1 * x ] in
   a[3]
```

Out of bounds access/update should be runtime errors

Extending the Snake Language

Diamondback: Arrays

```
def main(x):
  let a = [x , -1 * x ] in
   isArray(a)
```

support tag checking as with ints, bools

Extending the Snake Language

Diamondback: Arrays

```
def main(x):
  let list = [0, 1, false] in
  let _ = list[2] := list in
  ...
```

mutable updates allow for cyclic data

Concrete Syntax

```
‹expr›: ...
    | ‹array›
    | ‹expr› [ ‹expr› ]
    | ‹expr› [ ‹expr› ] := ‹expr›
    | newArray ( ‹expr› )
    | isBool ( ‹expr› )
    | isInt ( ‹expr› )
    | isArray ( ‹expr› )
    | length ( ‹expr› )
```

```
‹exprs›:
    | ‹expr›
    | ‹expr› , ‹exprs›
‹array›:
    | [ ]
    | [ ‹exprs› ]
```

Abstract Syntax

```
enum Prim {
  ...
  // Unary
  IsArray,
  IsBool,
  IsInt,
  NewArray,
  Length,

  MakeArray, // 0 or more arguments
  ArrayGet, // first arg is array, second is index
  ArraySet, // first arg is array, second is index, third is new value
}
```

Extending the Snake Language

Diamondback: Arrays

Semantics:

1. Each time we allocate an array should be a new memory location, so that updates don't overwrite previous allocations

2. What value does `e1[e2] := e3` produce?
   options: a constant, the value of e1 or e3, the old value of `e1[e2]`

3. Is equality of arrays by value or by reference?

   [0, 1, 2] == [0 , 1, 2]

Allocating Arrays

Where should the contents of our arrays be stored?

- Stack?

- Heap?

Stack Allocation

Can we allocate our arrays on the stack?

```
def main(x):
  let a = [x , -1 * x ] in
   a[1] := 0
```

Stack Allocation

Can we allocate our arrays on the stack?

```
def main(x):
  let a = [0, 1] in
  def f(n):
    a[n] + a[n + 1]
  in
  x + f(0)
```

Stack Allocation

Can we allocate our arrays on the stack?

```
def main(x):
 def f():
    [0, 1, 2, 3, 4]
 in
 def g(arr, i, j, k):
   arr[i] * arr[j] * arr[k]
 in
 let arr = f() in
 g(arr, 0, 2, 4)
```

If f allocates in its stack frame and returns a pointer,

The memory will be overwritten by any future calls

Doing this safely would require **copying** any returned data into the caller's stack frame. Not feasible for dynamically sized values.

Stack Allocation

Dynamically sized data can only feasibly be stack allocated if it is **local** to the function, i.e., only used in call stacks that contain the current function's stack frame.

If the dynamically sized data is **returned** from the function that allocates it, we instead allocate it in a separate memory region, the **heap**, and return a pointer to it.

Heap Allocation

The heap contains data whose lifetime is not tied to a local stack frame.

This makes the usage of the data more flexible, but complicates the question of when the data is **deallocated**.

For today, let's assume we do not deallocate memory.
  A strategy used in some specialized applications (missiles)

Today's simple heap model: the heap is a large region of memory, disjoint from the stack, some of it is used, and we have a pointer to the next available portion of memory.

The Heap

Let's take a particularly simple view of the heap for now: the heap is a large region of memory, disjoint from the stack. Some amount of this space is used, and we have a **heap pointer** that points to the next available region.

If memory is never deallocated (but also in copying gc), the structure is similar to the stack: we have a region of used space and a region of free space and the **heap pointer**, like the stack pointer, points to the beginning of the free space.

While the stack grows downward in memory, the heap grows upward.

Memory Management

Need our assembly programs to have access to the heap pointer at all times.

We will implement management of the heap in our **runtime system**, i.e., in Rust. Our assembly code programs will interface with the runtime system by calling functions the runtime system provides.

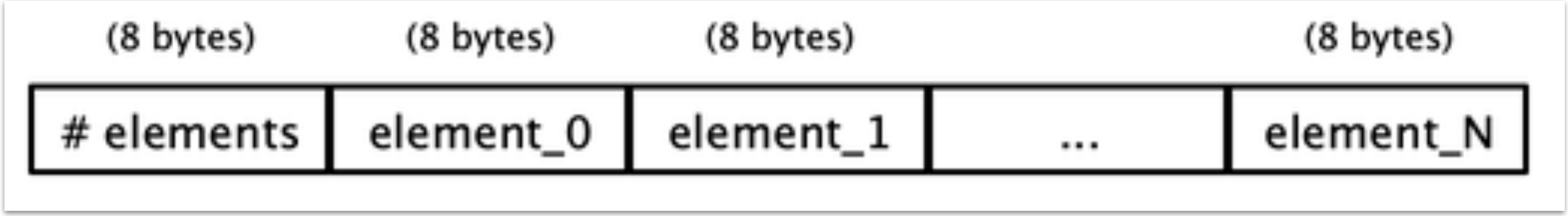
Implementing Arrays

When we implement arrays, we have two different representations to define:
1. How they are stored as "objects" in the heap
2. How they are represented as Snake values

Arrays as Objects

What data does an array need to store?

1. Need to layout the values sequentially so we can implement get/set
2. Need to store the **length** of the array to implement length as well as bounds checking for get/set.



Arrays as Values

The Snake value we store on the stack for an array is a **tagged** pointer to the array stored on the heap.

We overwrite the 2 least significant bits of the pointer with the tag 0b11.

This is safe, as long as those 2 least significant bits of the pointer contain no information, i.e., if they are always 0.

2 least significant bits of a pointer are 0 means the address is a multiple of 4, meaning the address is at a 4-byte alignment.

All arrays on our heap take up size that is a multiple of 8 bytes, so as long as the base of the heap is 4-byte aligned, we maintain this invariant.

Demo: Heap Sandbox

Summary:

Pre-allocate a large chunk of memory for our Snake program to use as its heap.

Allocation is managed by the runtime system, i.e., the stub.rs code.

Implementing Array Operations

Like with dynamically typed booleans, implementing array operations involves a combination of

1. Checking tags to ensure that the inputs are valid
2. Removing tags to get access to the underlying pointers
3. "Actual" loads and stores to memory
4. Adding tags to outputs

Implementing Array Operations

Like with dynamically typed booleans, implementing array operations involves a combination of

1. Checking tags to ensure that the inputs are valid

2. Removing tags to get access to the underlying pointers

3. "Actual" loads and stores to memory

4. Adding tags to outputs

As with booleans, we will add **assertions** as primitives to SSA, but implement the rest using new SSA operations for load/store.

SSA Extensions

1. **assertArray(x)**

   fail if x is not tagged as an array

2. **assertInBounds(n, m)**

   fail if m is an out of bounds index into a length n array, i.e., assert m < n

2. **load(p, off)**

   load 8 bytes of memory at [p + off * 8]

3. **store(p, off, v)**

   store the 8-byte value v at [p + off * 8]

4. **allocateArray(n)**

   allocate an array of length n from the runtime system

Implementing New Operations

1. **assertArray(x):** similar to assertInt, assertBool

2. **assertInBounds(n, m)**
```
cmp n, m
jle oob_error
```

3. **load(p, off)**
```
mov dest, [p + off * 8]
```

4. **store(p, off, v)**
```
mov [p + off * 8], v
```

5. **allocateArray(n):** call into the RTSs

Translation to SSA

1. newArray
2. array literals
3. array access
4. array update
5. isArray

Translation to SSA

Array allocation

Diamondback

```
newArray(e)
```

SSA

```
...
n = ... compile e
assertInt(n)
l = n >> 1
arr = allocateArray(n)
res = arr | 0b11
b
```

`Continuation:`
result stored in `res`
body of cont: **b**

Translation to SSA

Array literals

Diamondback

```
[e0 , ... , e(n−1)]
```

SSA

```
...
x0 = ... compile e0
...
arr = allocateArray(n)
store(arr, 1, x0)
...
store(arr, n, x(n−1))
res = arr | 0b11
b
```

`Continuation:`
result stored in `res`
body of cont: **b**

Translation to SSA

Array access

Diamondback

```
e1[e2]
```

`Continuation:`

result stored in `res`

body of cont: **b**

SSA

```
...
x1 = ... compile e1
...
x2 = ... compile e2
assertArray(x1)
assertInt(x2)
arr = x1 ^ 0b11
len = load(arr, 0)
ix = x2 >> 1
assertInBounds(len, ix)
ix2 = ix + 1  ; skip over the length
res = load(arr, ix2)
b
```

Translation to SSA

Array update

Diamondback

```
e1[e2] := e3
```

Continuation:
result stored in `res`
body of cont: **b**

SSA

```
...
x1 = ... compile e1
...
x2 = ... compile e2
...
x3 = ... compile e3
assertArray(x1)
assertInt(x2)
arr = x1 ^ 0b11
len = load(arr, 0)
ix = x2 >> 1
assertInBounds(len, ix)
ix2 = ix + 1  ; skip over the length
store(arr, ix2, x3)
res = x3
b
```

Translation to SSA

Array tag check

Diamondback

```
isArray(e)
```

```
Continuation:
```

result stored in `res`
body of cont: **b**

→

SSA

```
...
x = ... compile e
tag = x & 0b11
isArr = tag == 0b11
shifted = isArr << 2
res = shifted | 0b01
b
```

Array Summary

1. Extend runtime with a memory allocator, error functions

2. Extend translation to SSA to insert assertions, manipulate the runtime representation

3. Extend SSA to x86 to support loads, stores, assertion/allocator calls.