EECS 483: Compiler Construction

Lecture 12:

Arrays Continued, First-class Functions

February 24
Winter Semester 2025

Reminder

Assignment 3 (Procedures) due on Friday

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

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
}
```

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.

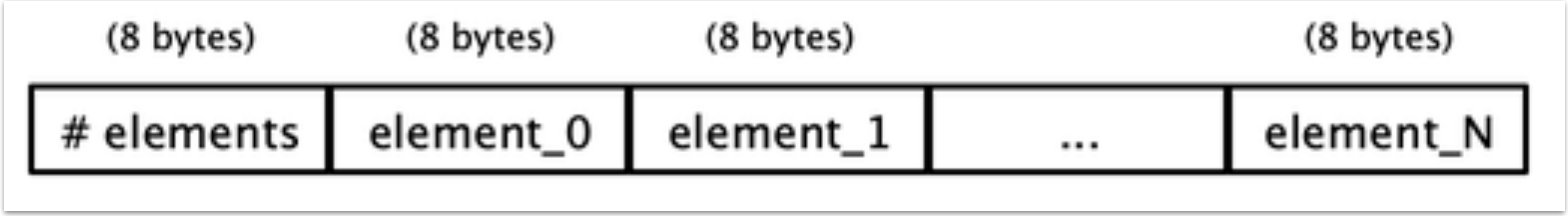
Implementing Arrays

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.

Heap/Runtime Demo

Live code: runtime system

Heap/Runtime Demo

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)
```

SSA

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

`Continuation:`

result stored in `res`

body of cont: **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.

Functions as Values

So far in our Snake language, functions are **second class,** meaning that unlike integers/booleans/arrays:

- ordinary program variables cannot be functions
- functions can't be passed as arguments to other functions
- functions can't be returned as values from other functions

This restriction simplifies the job of the compiler, but is uncommon in modern programming languages.

Functions as Values

Modern programming languages allow us to use functions as first-class data

- Low level languages like C/C++ have **function pointers**, which can be passed and returned like any other pointer type

- Higher-level languages both statically (C++, Rust, Java, Go, OCaml, Haskell) and dynamically typed (Python, Ruby, JavaScript, Racket) allow for a more flexible type called **closures**, sometimes called **lambdas**

Used as a convenient interface for implementing iterators, callbacks, concurrency,...

Functions as Values

```
def applyToFive(f):
  f(5)
in

def incr(x):
  x + 1
in

applyToFive(incr)
```

Functions as Values

```
def map(f, a):
  let l = length(a) in
  let a2 = newArray(l) in
  def loop(i):
    if i == l:
      true
    else:
      let _ = a2[i] := f(a[i]) in
      loop(i + 1)
  in
  let _ = loop(0) in
  a2
```

```
def incr(x): x + 1 in
def sqr(x): x * x in
let a = [0, 1, 2] in
map(incr, map(sqr, a))
```

Functions as Values

```
def map(f, a):
  let l = length(a) in
  let a2 = newArray(l) in
  def loop(i):
    if i == l:
      true
    else:
      let _ = a2[i] := f(a[i]) in
      loop(i + 1)
  in
  let _ = loop(0) in
  a2
```

```
let a = [0, 1, 2, 3] in
def sqr_a(i): a[i] := a[i] * a[i] in
let _ map(sqr_a, [1,3]) = in
a
```

need to support variable capture

Lambda Notation

```
def map(f, a):
  let l = length(a) in
  let a2 = newArray(l) in
  def loop(i):
    if i == l:
      true
    else:
      let _ = a2[i] := f(a[i]) in
      loop(i + 1)
  in
  let _ = loop(0) in
  a2
```

```
def incr(x): x + 1 in
def sqr(x): x * x in
let a = [0, 1, 2] in
map(incr, map(sqr, a))
```

Lambda Notation

```
def map(f, a):
  let l = length(a) in
  let a2 = newArray(l) in
  def loop(i):
    if i == l:
      true
    else:
      let _ = a2[i] := f(a[i]) in
      loop(i + 1)
  in
  let _ = loop(0) in
  a2
```

```
let a = [0, 1, 2] in
map(lambda x: x + 1 end,
    map(lambda x: x * x end,
    a))
```

Lambda Notation

Lambda notation is a syntax for defining function values directly rather than using `def`

```
lambda x1, x2,...: e end
```

Convenient for defining small functions to pass to map/filter/fold, etc.

Lambda Notation

Does lambda notation add any expressive power over local function definitions?

No.

```
lambda x1, x2,...: e end
```

⇒

```
def foo(x1, x2,...): e in
foo
```

Lambda Notation

What about the other way around? Are there functions we can define using local function definitions that we **can't** define using just lambda notation?

We can try a reverse translation:

```
def f(x1, x2,...): e1 in
e2
```

```
let f = lambda x1, x2,...: e1 in
e2
```

what goes wrong?

Lambda Notation

What about the other way around? Are there functions we can define using local function definitions that we **can't** define using just lambda notation?

We can try a reverse translation:

```
let fac = (lambda n:
  if n < 1: 1
  else: n * fac(n - 1))
in fac(5)
```

recursive call to fac is out of scope, because **let** bindings are not recursive

it is possible to desugar functions to just lambda (Y combinator), but harder to compile resulting code efficiently