EECS 483: Compiler Construction

Lecture 10:
Dynamic Typing

February 17
Winter Semester 2025

Announcements

- Assignment 3 released
  - extern functions, lambda lifting, SysVAMD64 calling convention
  - Start early!

State of the Snake Language

Adder: Straightline Code (arithmetic circuits)

Boa: local control flow (finite automata)

Cobra: procedures, extern (pushdown automata)

Remaining limitations:

1. Only data are ints (booleans are really just special ints)
2. Only ways to use memory are local variables and the call stack

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

Booleans in Boa/Cobra

In Boa/Cobra, booleans and integers weren't truly distinct datatypes.

- All integers could be used in logical operations
- All booleans could be used in arithmetic operations

Booleans in Boa/Cobra

The following are all valid programs with well-defined semantics in Boa/Cobra

```
-1 && 3

true + 5

7 >= false
```

Let's change the language semantics so these are **errors** instead.

Booleans in Boa/Cobra

Can we implement operations **isInt** and **isBool** that distinguish between integers and booleans?

```
isInt(true) == false

isInt(1)    == true
```

No, true and 1 have the exact same representation at runtime

Static vs Dynamic Typing

How would we implement a language where integers and booleans were considered disjoint?

1. Static Typing (C/C++, Java, Rust, OCaml)

   Identify the runtime types of all variables in the program

   Reject type-based misuse of values in the frontend of the compiler.

2. Dynamic Typing (JavaScript, Python, Ruby, Scheme)

   Use **type tags** to identify the type of data at runtime

   Reject type-based misuse of values at runtime, right before the operation is performed

Static Typing vs Dynamic Typing

Example 1:

```
    true + 5
```

Static typing: **compile time error**: true used where integer expected

Dynamic typing: **runtime error**: addition operation expects inputs to be integers

Static Typing vs Dynamic Typing

Example 2:

```
def main(x):
  x + 5
```

Static typing: need to declare a type for **x**, in this case **int**

Dynamic typing: succeed at runtime if **x** is an int, otherwise fail

Static Typing vs Dynamic Typing

Example 2:

```
def main(a):
  def complex_function(): ... in
  let x = if complex_function(): 1 else: true
  x + 5
```

Static typing: reject this program, even if **complex_function** always returns true

Dynamic typing: succeed at runtime if **complex_function** returns true, otherwise fail

Static Typing vs Dynamic Typing

Static Typing

- Easier on the compiler: if type information is reliable, we can use that to inform the runtime representation of our compiled values
- Easier on the programmer? Types document the code, aid in tooling, design

Dynamic Typing

- Easier on the programmer? Complex patterns that are difficult to assign static types are possible

In this class we are value neutral: both are common, so we study both. In Assignment 4 we will implement dynamic typing, and

Static Typing vs Dynamic Typing

Poll: Is static typing or dynamic typing better?

My opinion:

> I prefer static typing, but both are popular enough to be worth studying and implementing well.

In Assignment 4, we'll implement dynamic typing

In Assignment 5, perform optimizations to reduce the runtime overhead of dynamic typing

Revisit syntactic aspects of static typing and the relation with static analysis later in the course.

Semantics of Dynamic Typing

Live code interpreter

Semantics of Dynamic Typing

- A Snake value is not just an int anymore. It is **either** an int or a boolean, and we need to be able to tell the difference at runtime in order to determine when we should error and how to implement isInt, isBool.
- Many operations can now produce runtime errors if type tags are incorrect, need to specify
  - what the appropriate error messages are
  - evaluation order between expressions executing and type tags
    - true + (let _ = print(3) in 3)
      - does this print 3 before it errrors?

Representing Dynamically Typed Values

In Adder/Boa/Cobra, all runtime values were integers.

In Diamondback, a runtime value must have both a type tag and a value that matches the type tag

How should we represent tags and values in our compiled program?

Representing Dynamically Typed Values

Approach 1: Values as 8 bytes, Tags as extra data

A snake value is 9 bytes

- the first byte is a tag: 0x00 for integer, 0x01 for boolean. Use a full byte to keep our values byte-aligned
- the remaining 64 bits are the underlying integer, bool or pointer

Upside: Faithful representation of our Rust interpreter

Downside: 1 byte memory overhead for all values plus padding, calling convention and architecture are 8-byte oriented, tedious to implement pervasively

Representing Dynamically Typed Values

Approach 2: Values as pointers

- A snake value is a 64-bit pointer to an object on the **heap**
  - value stored on the heap can then be whatever size we want, the pointer is always 64 bits.
  - store a tag and value on the heap similarly to previous approach.

A value stored in this way is called **boxed**.

Representing Dynamically Typed Values

Approach 2: Values as pointers

- A snake value is a 64-bit pointer to an object on the **heap**
  - value stored on the heap can then be whatever size we want, the pointer is always 64 bits.
  - store a tag and value on the heap similarly to previous approach.

Upside: uniform implementation, 64-bit values can be compiled as before

Downside: memory overhead. Accessing the tag requires a non-local memory access, performing an arithmetic operation multiple

Approach taken in Python

Representing Dynamically Typed Values

Approach 3: compromise

A snake value is a 64-bit value.

- Use the least significant bits of the value as a **tag**.
- Represent simple data like integers, booleans within the 64-bits
- Represent large datatypes like arrays, closures, structs as pointers to the heap

Upside: use stack allocation more often

Downside: can't fit 64 bits and a tag...

Roughly the approach used in high-performance Javascript engines (v8) as well as some garbage-collected typed languages (OCaml)

Representing Dynamically Typed Values

To implement our compiler, we need to specify

1. How each of our Snake values are represented at runtime
2. How to implement the primitive operations on these representations

Integers

Implement a snake integer as a 63-bit signed integer followed by a 0 bit to indicate that the value is an integer

Number	Representation
1	`0b00000000_0000.....0000_00000010`
6	`0b00000000_0000.....0000_00001100`
-1	`0b11111111_1111.....1111_11111110`

I.e., represent a 63-bit integer **n** as the 64-bit integer 2 * n

Booleans

The least significant bit must be 1 to distinguish from integers

Use least significant bits `0b01` to distinuish from integers and other datatypes

Use the remaining 62 bits to encode true and false as before as 1 and 0

Number	Representation
`true`	`0b00000000_0000.....0000_00000101`
`false`	`0b00000000_0000.....0000_00000001`

2^62 - 2 bit patterns are therefore "junk" in this format

Boxed Data

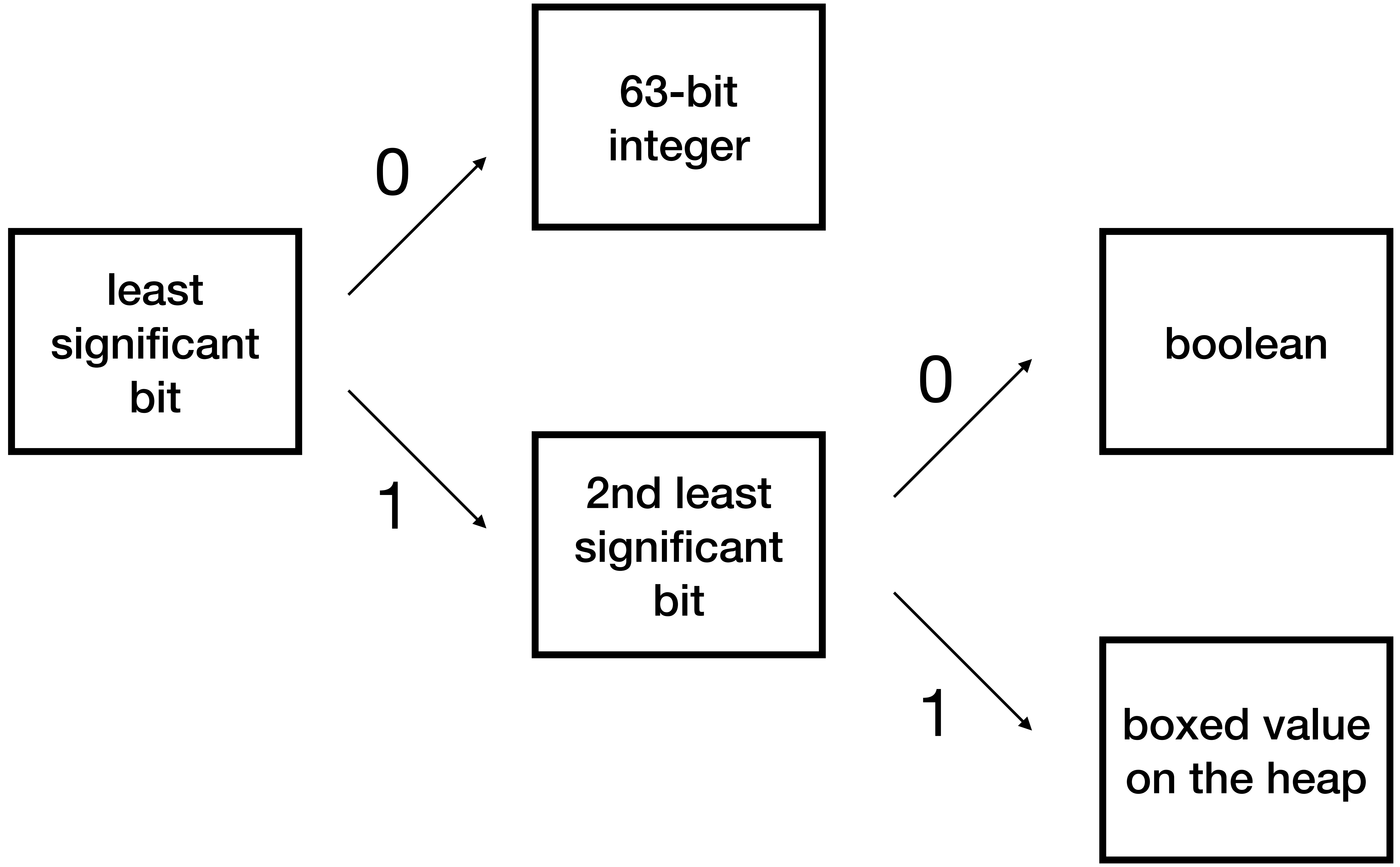
The least significant bit must be 1 to distinguish from integers

Use least significant bits `0b11` to distinguish from booleans.

Use remaining 62-bits to encode a pointer to the data on the heap

Why is this ok? Discuss more thoroughly on Wednesday

Representing Dynamically Typed Values



Implementing Dynamically Typed Operations

We need to revisit our implementation of all primitives in assembly code to see how they should work with our new datatype representations.

1. Arithmetic operations (add, sub, mul)
2. Inequality operations (<=, <, >=, >)
3. Equality
4. Logical operations (&&, ||, !)

As well as supporting our new operations isInt and isBool

Implementing Dynamically Typed Operations

In dynamic typing, implementing a primitive operation has two parts:

1. How to check that the inputs have the correct type tag
2. How to actually perform the operation on the encoded data

Implementing Dynamically Typed Operations

Live code

Compiling Dynamic Typing

We know what the source semantics is and what kind of assembly code we want to generate.

In implementing the compiler, we now we have a design choice: in what phase of the compiler do we actually "implement" dynamic typing?

1. Implement everything in x86 code generation
2. Implement everything in lowering to SSA
3. Implement in multiple passes

Compiling Dynamic Typing

Approach 1: implement all dynamic typing semantics in code generation.

In this case, SSA values would be dynamically typed, like Diamondback

Compiling Dynamic Typing

Approach 1: implement all dynamic typing semantics in code generation.

Diamondback

```
x + y
```

SSA

```
r = x + y
ret r
```

x86

```
mov rax, [rsp - 8]
test rax, 1
jnz err_arith_exp_int
mov r10, [rsp - 8]
test r10, 1
jnz err_arith_exp_int
add rax, r10
ret
```

Compiling Dynamic Typing

Approach 1: implement all dynamic typing semantics in code generation.

In this case, SSA values would be dynamically typed, like Diamondback

Downside: goes against the philosophy that SSA should be thin wrapper around the assembly code.

Makes the semantics of SSA more complex and so the code generation more complex.

More complex code generation: missed opportunities for SSA-based optimization

Compiling Dynamic Typing

Approach 2: implement dynamic typing in the translation to SSA

In this approach, SSA values are as before always 64-bit integers, and SSA operations work on these 64-bit integers (as they do now)

Compiling Dynamic Typing

Approach 2: implement dynamic typing in the translation to SSA

Diamondback

```
x * y
```

SSA

```
check_y():
  y_bit = y & 1
  c = y_bit == 0
  cbr mult_xy() err()
mult_xy():
  tmp = x * y
  z = tmp >> 1
  ret z
x_bit = x & 1
b = x_bit == 0
cbr check_y() err()
...
```

Compiling Dynamic Typing

Approach 2: implement dynamic typing in the translation to SSA

Benefit: code generation is very simple, at cost of SSA lowering more complex

Downside: difficult to optimize unnecessary tag checks away

Compiling Dynamic Typing

Approach 3: implement dynamic typing in multiple passes

In lowering to SSA, make some aspects of dynamic typing explicit but leave the tag checking as primitive operations.

Implement the tag checking in the x86 code generation.

Compiling Dynamic Typing

Approach 3: implement **some** dynamic typing in SSA lowering

Diamondback

```
x * y
```

SSA

```
assertInt(x)
assertInt(y)
r = x * y
s = r >> 1
ret s
```

Insert type tag assertions in SSA, implement bit-twiddling manually

Compiling Dynamic Typing

Approach 3: implement **some** dynamic typing in SSA lowering

SSA

```
assertInt(x)
```

x86

```
mov rax, [rsp - offset(x)]
test rax, 1
jnz assert_int_fail
...
assert_int_fail:
  sub rsp, 8
  call snake_assert_int_error
```

Compiling Dynamic Typing

Optimization opportunity

Diamondback

```
def fact(x):
  if x == 0:
    1
  else:
    x * fact(x - 1)
in
fact(7)
```

SSA

```
...
tmp1 = x - 1
tmp2 = call fact(tmp1)
assertInt(x)
assertInt(tmp2)
r = x * tmp2
ret r
```

will these **assertInt** ever fail?

Compiling Dynamic Typing

Optimization opportunity

Diamondback

```
def fact(x):
  if x == 0:
    1
  else:
    x * fact(x - 1)
in
fact(7)
```

SSA

```
...
tmp1 = x - 1
tmp2 = call fact(tmp1)
```
~~`assertInt(x)`~~
~~`assertInt(tmp2)`~~
```
r = x * tmp2
ret r
```

with a simple **static analysis** determine that x, tmp2 are always integers. Remove unnecessary assertions

Compiling Dynamic Typing

Compare to approach 2:

Diamondback

```
x * y
```

how would we remove the checking from the code on the right?

SSA

```
check_y():
  y_bit = y & 1
  c = y_bit == 0
  cbr mult_xy() err()
mult_xy():
  tmp = x * y
  z = tmp >> 1
  ret z
x_bit = x & 1
b = x_bit == 0
cbr check_y() err()
...
```

Summary: Adding Dynamic typing

How does adding dynamic typing affect each pass of our compiler?

Changes to Frontend

New error: only 63-bit integers are supported, so need to reject 64-bit values in the parser/frontend

Changes to Middle End

Diamondback values: tagged data, either a 63-bit int, or true or false

SSA values: 64-bit integers

Add primitive assertions **assertInt** and **assertBool** to SSA

Assume after an assertion executes that it passed

Use bitwise operation in SSA to encode the Diamondback values/ operations

Changes to Back End

Implement **assertInt** and **assertBool** operations in x86, calling out to functions implemented in Rust to display appropriate errors

Changes to Runtime (stub.rs)

Parse input arguments into snake values.

Update printing to account for new representation

Implement functions that display runtime errors and exit the process