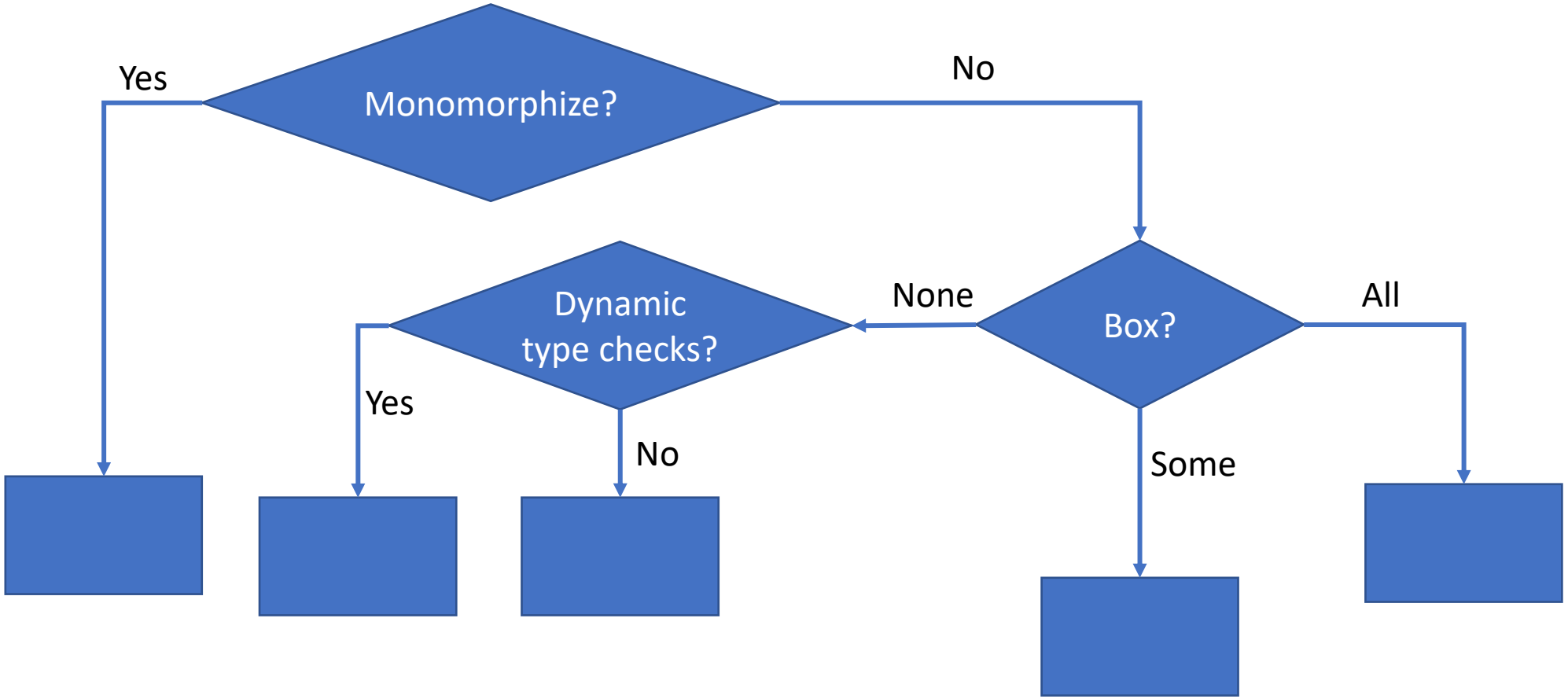
CS443: Compiler Construction

Lecture 12: Compiling ML Values

Stefan Muller

There are a lot of ways to compile values



We (probably) want a uniform representation of values

```
'a list
```

```
struct __list{
  value hd;
  __list tl;
};
```

Could pull the "pick a default type and cast as necessary" trick but still want values to all be the same size

First option: actually just have one type of values

```
enum Tag {INT, BOOLEAN, …} ;

struct Int { enum Tag t ; int value ; } ;

struct Boolean { enum Tag t ; unsigned int value ; } ;
…
union Value {
  enum Tag t ;
  struct Int z ;
  struct Boolean b ;
  …
} ;
```

Courtesy Matt Might: https://matt.might.net/articles/compiling-scheme-to-c/

Then we have to check the tag of an object when we use it…

```
Value neg(Value i) {
  switch (i.t) {
    case INT:
      Int ret;
      ret.t = INT;
      ret.value = -((Int) i).value;
      return ret;
    default:
      //Type Error!
      exit 1;
  }
}
```

Easy, Slow, Wasteful

…or do we?

- No (in a statically typed language without something like `instanceof`)

```
Value neg(Value i) {
  Int ret;
  ret.t = INT;
  ret.value = -((Int) i).value;
  return ret;
}
```

Easy, Fast, Wasteful

Second option: “Boxing” (use pointers for everything)

```
typedef void * Value

struct Int { int value; };
struct Boolean { bool value; };
struct List { Value hd; Value tl };
```

Key idea: we may not know a value’s value at compile time, but we know its type!

Second option: “Boxing” (use pointers for everything)

```
let l: int list = 1::[]
in (hd l) + 2
```

```
Value l = malloc(sizeof(List));
Value i = malloc(sizeof(Int));
((Int *)i)->value = 1;
((List *)l)->hd = i;
((List *)l)->tl = null;
Value i2 = malloc(sizeof(Int));
((Int *)i2)->value = 2;
return ((Int *)((List *) l)->hd)->value + ((Int *) i2)->value
```

Harder, slower, still pretty wasteful

Compromise: “Unbox” ints, other small base types

```
let l: int list = 1::[]
in (hd l) + 2
```

```
Value l = malloc(sizeof(List));
((List *)l)->hd = ((Value) 1);
((List *)l)->tl = null;
return ((int) ((List *)l)->hd) + 2
```

Harder, relatively fast, space-efficient

Have structs for different types

```
struct __list {
  int list_hd;
  __list list_tl;
};

struct __pair {
  int pair_fst;
  int pair_snd;
};
```

We need to pick a default type for values.
May as well use int (no void* in MiniC)

We still need dynamic tag checks for ADTs

```
type exp = EVar of string
         | EBinop of exp * exp
```

```
enum exp_tag { EVAR; EBINOP };
union exp;
struct EVar {
  exp_tag t;
  char[] arg1;
}
struct EBinop {
  exp_tag t;
  union exp *arg1;
  union exp *arg2;
}
union exp {
  struct EVar evar;
  struct EBinop ebinop;
}
```

A totally different option: get rid of polymorphism (“monomorphize”)

```
struct int_list{
  int hd;
  __list tl;
};

struct bool_list{
  bool hd;
  __list tl;
};
```

That means we need to make different versions of polymorphic functions

```
let pair (x: ‘a) : ‘a * ‘a = (x, x)
```

```
intpair pair_int(x: int) { … }
boolpair pair_bool(x: bool) { … }
```

…

(We’ll also need pair_intpair, pair_intboolpair, …)

To monomorphize functions, we need to know all the ways they can be used

- Check all call sites ➔ Whole program compilation

Much harder
Slow, non-modular compilation
Blindingly fast at runtime
Space-efficient

There are a lot of ways to compile values