Types and Systems Programming

Advanced Systems Programming (H)

Lecture 3

Lecture Outline

- Strongly Typed Languages
  - What is a strongly typed language?
  - Why is strong typing desirable?
  - Types for systems programming
- Introducing the Rust programming language
  - Basic operations and types
  - Pattern matching
  - Memory management
  - Why is Rust interesting?

Strongly Typed Languages

- What is a strongly typed language?
- Why is strong typing desirable?
- Types for systems programming

What is a Type?

- A type describes *what* an item of data represents
  - Is it an integer? floating point value? file? sequence number? username?
  - What, conceptually, is the data?
  - How is it represented?
- Types are very familiar in programming

```
int    x;
double y;
char  *hello = “Hello, world”;

struct sockaddr_in {
  uint8_t         sin_len;
  sa_family_t     sin_family;
  in_port_t       sin_port;
  struct in_addr  sin_addr;
  char            sin_pad[16];
};
```

Declaring variables and specifying their type

Declaring a new type, `struct sockaddr_in`

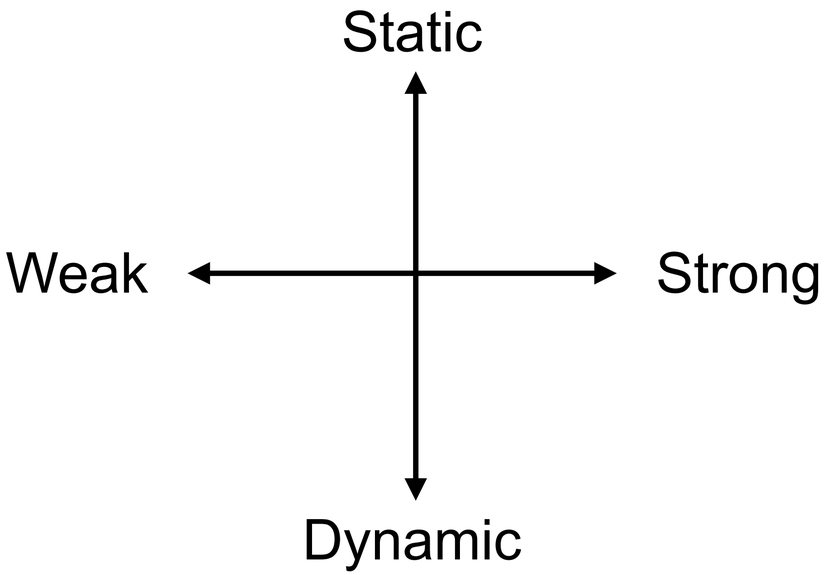
What is a Type System? (1/2)

- A type system is a set of rules constraining how types can be used
  - What operations can be performed on a type?
  - What operations can be performed with a type?
  - How does a type compose with other types of data?

What is a Type System? (2/2)

- A type system proves the absence of certain program behaviours
  - It doesn’t guarantee the program is correct
  - It does guarantee that *some* incorrect behaviours do not occur
    - A good type system eliminates common classes of bug, without adding too much complexity
    - A bad type system adds complexity to the language, but doesn't prevent many bugs
  - Type-related checks can happen at compile time, at run time, or both
    - e.g., array bounds checks are a property of an array type, checked at run time

Types of Type Systems



- Can objects changes their type?
- How strictly are the typing rules enforced?

Static and Dynamic Types (1/3)

- In a language with static types, the type of a variable is fixed:
  - Some require types to be explicitly declared; others can infer types from context
  - Just because the language can infer the type does not mean the type is dynamic:

```
> cat src/main.rs
fn main() {
    let x = 6;
    x += 4.2;
    println!("{}", x);
}
> cargo build
   Compiling hello v0.1.0 (/Users/csp/tmp/hello)
error[E0277]: cannot add-assign `{float}` to `{integer}`
 --> src/main.rs:3:7
  |
3 |     x += 4.2;
  |       ^^ no implementation for `{integer} += {float}`
  |
  = help: the trait `std::ops::AddAssign<{float}>` is not implemented for `{integer}`

error: aborting due to previous error
```

The Rust compiler infers that `x` is an integer and won't let us add a floating point value to it, since that would require changing its type

Static and Dynamic Types (2/3)

- In a language with dynamic types, the type of a variable can change:

```
> python3
Python 3.6.2 (v3.6.2:5fd33b5926, Jul 16 2017, 20:11:06)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> x = 6
>>> type(x)
<class 'int'>
>>> x += 4.2
>>> type(x)
<class 'float'>
>>>
```

Static and Dynamic Types (3/3)

- Dynamically typed languages tend to be lower performance, but offer more flexibility
  - They have to store the type as well as its value, which takes additional memory
  - They can make fewer optimisation based on the type of a variable, since that type can change
- Systems languages generally have static types, and be compiled ahead of time, since they tend to be performance sensitive

Strong and Weak Types (1/2)

- In a language with **strong** types, every operation must conform to the type system
  - Operations that cannot be proved to conform to the typing rules are not permitted
- **Weakly** typed languages provide ways of circumventing the type checker:
  - This might be automatic safe conversions between types:

```
float  x = 6.0;
double y = 5.0;
double z = x + y;
```

  - Or it might be an open-ended cast:

```
char *buffer[BUFLEN];
int   fd = socket(…);
…
if (recv(fd, buffer, BUFLEN, 0) > 0) {
  struct rtp_packet *p = (struct rtp_packet *) buf;
  …
}
```

Common C programming idiom: casting between types using pointers to evade the type system

Strong and Weak Types (2/2)

- Think of **strong** and **weak** types in the context of **safe** and **unsafe** languages:
  - A **safe** language, whether static or dynamic, know the types of all variables and only allows legal operations on those values
  - An **unsafe** language allows the types to be circumvented to perform operations the programmer believes correct, but the type system can't prove to be so

Why is Strong Typing Desirable?

- Results of a program using only **strong types** are well-defined → a **safe** language
  - Type system ensures results are consistent with the rules of the language
  - A strongly-typed program will only ever perform operations on a type that are legal – cannot perform undefined behaviour

- Use of **strong types** helps model the problem, check for consistency, and eliminate common classes of bug
- Strong typing **cannot** prove that a program is correct, but **can** prove the absence of certain classes of bug

`Segmentation fault (core dumped)`

Segmentation faults should never happen:

- Compiler and runtime should strongly enforce type rules
- If program violates them, it should be terminated cleanly
- Security vulnerabilities come from undefined behaviour after type violations

Segmentation fault (core dumped)

3.4.3

1 **undefined behavior**

behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

2 NOTE Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

C has 193 kinds of undefined behaviour

Appendix J of the C standard https://www.iso.org/standard/74528.html ($) or
http://www.open-std.org/jtc1/sc22/wg14/www/abq/c17_updated_proposed_fdis.pdf

Undefined behaviour leads to *entirely unpredictable* results → https://blog.regehr.org/archives/213

Types for Systems Programming

- C is weakly typed and widely used for systems programming
  - Why is this?
  - Can systems programming languages be strongly typed?
  - What are the challenges in strongly typed systems programming?

Why is C Weakly Typed?

- Mostly, historical reasons:
  - The original designers of C were not type theorists
  - The original machines on which C was developed didn't have the resources to perform complex type checks
  - Type theory was not particularly advanced in the early 1970s

Is Strongly-typed Systems Programming Feasible?

- Yes – many examples of operating systems written in strongly-typed languages
  - Old versions of macOS written in Pascal
  - Project Oberon http://www.projectoberon.com
  - US DoD and the Ada programming language
    - Aerospace, military, air traffic control
- Popularity of Unix and C has led to a belief that operating systems require unsafe code
  - True only at the very lowest levels
  - Most systems code, including device drivers, can be written in strongly typed, safe, languages
  - **Rust is a modern attempt to provide a type-safe language suited to systems programming**

```
type ErrorType   is range 0..15;
type UnitSelType is range 0..7;
type ResType     is range 0..7;
type DevFunc     is range 0..3;
type Flag        is (Set, NotSet);
type ControlRegister is
record
    errors    : ErrorType;
    busy      : Flag;
    unitSel   : UnitSelType;
    done      : Flag;
    irqEnable : Flag;
    reserved  : ResType;
    devFunc   : DevFunc;
    devEnable : Flag;
end record;

for ControlRegister use
record
    errors    at 0*Word range 12..15;
    busy      at 0*Word range 11..11;
    unitSel   at 0*Word range  8..10;
    done      at 0*Word range  7.. 7;
    irqEnable at 0*Word range  6.. 6;
    reserved  at 0*Word range  3.. 5;
    devFunc   at 0*word range  1.. 2;
    devEnable at 0*Word range  0.. 0;
end record;

for ControlRegister'Size use 16;
for ControlRegister'Alignment use Word;
for ControlRegister'Bit_order use Low_Order_First;
...
```

Strongly Typed Languages

- What is a strongly typed language?
- Why is strong typing desirable?
- Types for systems programming