Rust: Types for Aliasing Control

CS242

Lecture 11

Review

- Object-Oriented vs Functional Languages
  - Untyped lambda calculus is fine for expressing untyped object calculi

- But typed languages are a different story
  - Difficult to type method override without very strong restrictions on OO languages
  - Because method override can radically change the type of a program, difficult to define a natural translation from a typed OO language into a typed lambda calculus

Practical Impact

- There is no (known) best way of combining OO and functional features in a typed language
- OO core + functional features (Java, C++)
- Functional core + OO features (OCaml, Haskell)
- Go with a dynamically typed language (Python, Javascript)
  - Dynamically typed languages noticeably more popular in OO programming

Today’s Topics

- Motivation: Memory safety
- Aliasing
  - Classical approaches to aliasing control
- Rust
  - Type-based aliasing control in a practical language

Memory Safety

- Memory safety is the property that pointers or references point to objects of the correct type

- Memory safety bugs plague systems written in languages with manual memory management
  - Double-frees, wild pointers, and out-of-bounds accesses
  - Primarily C/C++
  - Major source of security vulnerabilities

Example

```
int *foo(int v) {
  int *ptr = (int *) malloc(sizeof(int));
  int err = initialize_int(ptr,v);
  if (err != 0) free(ptr);
  return ptr;
}
```

Example

```
int *foo(int v) {
   int *ptr = (int *) malloc(sizeof(int));
   int err = initialize_int(ptr,v);
   if (err != 0) free(ptr);
   return ptr;
}
```

```
void bar() {
   int *p = foo(42);
   … *p …            // wild pointer
   …
   free(p);          // double free
   …
}
```

How Can Memory Safety Be Assured?

- Three options:

- Automatically via dynamic garbage collection
- Systematic but unenforced programming disciplines
- Automatically via a static type system

Garbage Collection (GC)

- Three key properties
  - Deallocation is done automatically, not by the programmer
    - Many versions, all exploit: *objects that will never be used again are safe to deallocate*
  - No pointer arithmetic allowed
    - A *reference* is a pointer without pointer arithmetic
    - Guarantees the program cannot compute a pointer that GC doesn't know about
  - Indexing into arrays is bounds-checked

- Upside: Memory safe!

- Downside is performance costs of various kinds:
  - Bounds checks are expensive
  - Often inefficient for applications where the working set is a large fraction of memory
  - Unpredictable delays for GC

Who Deallocates?

Consider a function call:

```
void my_func() {
    int *ptr = (int *) malloc(sizeof(int));
    *ptr = 42;
    api_call(ptr);
    …
}
```

- Both my_func and api_call hold pointers to the integer
- Which is responsible for deallocating the memory?

The Ownership Programming Discipline

- Designers of large systems have always needed to talk about the system's rules for memory management
  - In particular, who is responsible for deallocating memory

- The *ownership* discipline is the most popular approach
  - One pointer is considered the *owner* of an allocated block of memory
  - The owner, and only the owner, is responsible for deallocating the block
  - Since every block has a unique owner, the risk of memory management errors is greatly reduced

Back to the Example …

Consider a function call:

```
void my_func() {
    int *ptr = (int *) malloc(sizeof(int));
    *ptr = 42;
    api_call(ptr);
    …
}

api_call(int *p) { … }
```

- Who is the owner, ptr or p?
- Answer: It depends, and the answer is different in different circumstances
- But ownership at least gives terminology for discussing desired memory management policies

Back to the Example …

Consider a function call:

```
void my_func() {
    int *ptr = (int *) malloc(sizeof(int));
    *ptr = 42;
    api_call(ptr);
    … more code …
}

api_call(int *p) { … }
```

- Last use of ptr is in "more code"
  - ptr should be the owner
- Last use is in api_call
  - p could be the owner
- api_call stores a pointer p' to the memory in a global data structure
  - p' should be the owner

Ownership Programming Discipline

- Each allocated object/memory block has a unique owner

- Ownership rules for a given system often documented in comments
  - E.g., for each pointer passed to an API

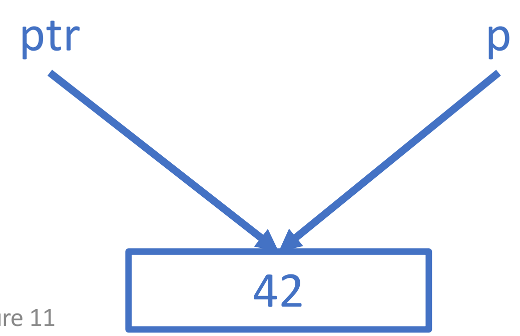
- But nothing enforces correct use
  - It is up to programmers to understand and respect the rules laid down for a specific system

A Key Concept: Aliasing

```
void my_func() {
    int *ptr = (int *) malloc(sizeof(int));
    *ptr = 42;
    api_call(ptr);
    …
}

api_call(int *p) { … }
```

- Notice that ptr and p are two different names for the same memory location
- We say ptr and p are *aliases*



A Key Concept: Aliasing

```
void my_func() {
    int *ptr = (int *) malloc(sizeof(int));
    *ptr = 42;
    api_call(ptr);
    …
}

api_call(int *p) { … }
```

- The modern view is that aliasing is a core issue
  - For memory safety and other things
- When trying to understand a piece of code with a pointer p, we generally do not know:
  - Are there aliases of p?
  - How long do aliases exist - do their lifetimes overlap with p?
  - Are aliases of p read, written or deallocated?

Aliasing Control

A Classic Example

```
copy(char *x, char  *y) {
      ...
}
```

But what about copy(a,a)?

A Classic Example

```
copy(restrict char *x, restrict char  *y) {
      ...
}
```

Semantics: In C, a restricted pointer cannot be aliased to any other pointer in scope.

A Point of View

- Aliasing is bad

- State can be modified through one name and those changes are visible through a different name
  - Leads to subtle and difficult bugs

- But aliasing is very common in real programs
  - Impossible to avoid
  - E.g., references passed as arguments to functions
  - Object-oriented code is particularly prone to generating aliasing

Idea #1

- Maybe aliasing is not the problem ...

- Problems arise only when aliasing is combined with mutation
  - That is, the ability to write/update state

- So, disallow mutation!
  - Can’t get surprises from aliases if only reads are allowed
  - The pure functional programming viewpoint

Could Outlawing Mutation Really Work?

- People have studied pure functional languages for decades
  - No mutation, whenever a data structure is changed a copy is made
- A surprising number of computational problems have very efficient algorithms without mutation of state
  - Sometimes just amortized bounds, but that is still quite good!
- But there are some operations that seem to fundamentally require mutation to be efficient
  - Update in place of an array is O(1)
  - The best known functional update is O(log N) in the size of the array

A Practical Approach

- Split the world into mutable and immutable values

- Rust
  - let x = 5 // immutable
  - let mut x = 5 // mutable
  - x = 3 // only allowed if x is mutable

- ML
  - let x = 5 // immutable
  - let x = ref 5 // mutable
  - x := 3

Separating Mutable & Immutable

- Not entirely a new idea
  - E.g., const in C

- Gaining in popularity
  - More languages are making this distinction
  - With immutability being the default

- Now accepted as a good idea
  - Limit the possibility of mutation to places it is really needed
  - Make these points obvious in the syntax & types

Idea #2

- Control aliasing in the type system
  - Track it, restrict it, or even disallow it

- Ownership types
  - Track aliases using types
  - Upgrades the ownership programming discipline to an enforced type discipline

- There is a large literature on ownership types
  - Some quite elaborate ...

Ownership in Rust

- Rust is the first widely used programming language with ownership

- There is always a single *owner* reference of every object
  - Owning = responsible for the resources of the object

- Implications
  - An object with no owner is deallocated
    - When an owner goes out of scope, the owned object is deallocated
  - Copies transfer ownership
    - x = y removes ownership from y and transfers it to x
    - y can no longer be used after the assignment

Ownership Example

```
fn main() {
   let v = vec[1,2,3];      // v owns the vector
   let v2 = v;              // moves ownership to v2
   display(v2);             // ownership is moved to display
}

fn display(v:Vec<i32>){
    println!("{:?}",v);
    // v goes out of scope here and the vector is deallocated
}
```

Ownership Example

```
fn main(){
   let v = vec[1,2,3];     // v owns the vector
   let v2 = v;             // moves ownership to v2
   let i = v[1];  compile-time error!
   display(v2);            // ownership is moved to display
   println!("{:?}",v2);  compile-time error!
}

fn display(v:Vec<i32>){
    println!("{:?}",v);
    // v goes out of scope here and the vector is deallocated
}
```

Another Ownership Example

```
fn a() {
let x = Foo::new();    // x is the owner
let y = bar(x);        // ownership is transferred to the argument of bar
                       //  and then back to y
// y goes out of scope and the Foo object is deallocated
}


fn bar(z: Foo) -> Foo {
   z;     // ownership is transferred back to the caller
}
```

Lifetimes

- Rust reasons about aliasing/ownership by using *lifetimes*
- The lifetime of a variable is the span between
  - The definition (first use)
  - The last use
- Rule: Lifetimes of the owners of an object cannot overlap

Lifetimes

```
fn main(){
    let v = vec[1,2,3];     // vector v owns the object
    let v2 = v;             // moves ownership to v2
    // let i = v[1];    compile-time error!
    display(v2);            // ownership is moved to display
}
```

v's lifetime

v2's lifetime

```
fn display(v:Vec<i32>){
    println!("{:?}",v);
    // v goes out of scope here and the vector is deallocated
}
```

v's lifetime

Lifetimes: A Compile Time Error

```
fn main(){
   let v = vec[1,2,3];     // vector v owns the object
   let v2 = v;             // moves ownership to v2
   let i = v[1];           // compile-time error!
   display(v2);            // ownership is moved to display
}

fn display(v:Vec<i32>){
    println!("{:?}",v);
    // v goes out of scope here and the vector is deallocated
}
```

Lifetimes: A Fix

```
fn main(){
  let v = vec[1,2,3];     // vector v owns the object
  let i = v[1];           // now this works …
  let v2 = v;             // moves ownership to v2
  display(v2);            // ownership is moved to display
}

fn display(v:Vec<i32>){
   println!("{:?}",v);
   // v goes out of scope here and the vector is deallocated
}
```

Another View

```
fn main() {
   let v = vec[1,2,3];     // v owns the vector
   let v2 = v;             // moves ownership
   display(v2);            // moves ownership
}

fn display(v:Vec<i32>){
    println!("{:?}",v);
    // v is deallocated
}
```

- Recall: Lifetimes of owners cannot overlap

- Enforces a *linear type* discipline
  - Only one name for an object is available at any time
  - Alternatively, guarantees no aliases are simultaneously available
  - No aliases => no problems with aliasing!

- Linear type systems have received a lot of attention
  - But linearity is a *very* strong restriction …

Aliasing Control in Rust

- Disallowing simultaneously available aliases is painful in many situations
  - Can never have a second name for an object or even a piece of an object
  - E.g., makes it impossible to write an array iterator
    - Need a name for the array and a pointer into the middle of the array
  - And we often don't need to take ownership anyway
    - Most aliases are temporary and used in controlled ways

- Rust allows the creation of explicit aliases
  - called *borrows*

- There are two kinds of borrows:
  - mutable
  - immutable

Example: Immutable Borrow

```
fn a() {
let x = Foo::new();          // x is the owner
let y = &x;                  // y is an immutable borrow of x;  x is still the owner
bar(y);                      // pass an immutable borrow to bar
}


fn bar(&z: Foo) {
  ... =  .. z ...            // can read from z in bar as many times as we like
  //     let global.f = z    storing z somewhere that outlives bar gives a type error
}
```

Example: Immutable Borrow

```
fn a() {
let x = Foo::new();   // x is the owner
let y = &x;           // y is an  immutable borrow of x;  x is still the owner
bar(y, y);            // pass two immutable borrows to bar
}


fn bar(a: &Foo, b: &Foo) {
  ... =  .. a ... // can read from a and b in bar as many times as we like
  ... = ... b ...

}
```

Example: Mutable Reference

```
fn a() {
x = Foo::new();      // x is the owner
y = &mut x;          //  y is a mutable borrow of x
bar(y);              // pass a mutable borrow to bar
}


fn bar(z: &mut Foo) {
  z.f =  ... // can mutate z
}
```

Example: Mutable Borrow

```
fn a() {
let x = Foo::new();      // x is the owner
let y = &mut x;          //  y is a mutable borrow of x
bar(y, y)                // Error: Cannot pass two mutable borrows of x to bar
}


fn bar(a: &mut Foo, b: &mut Foo) {   // since a and b are mutable, they cannot alias
  a.f =  ... // can mutate a
  b.f = ...  // can mutate b
}
```

Borrow Rules

- A borrow cannot outlive its owner
  - The lifetime of a borrow is contained within the lifetime of its owner
    - Guarantees no dangling references
- A borrow cannot deallocate its object
  - That's what it means to be a borrow and not the unique owner
- There can be one mutable borrow to an object in scope
  - There can be any number of immutable borrows
  - We relax the linearity restriction to allow any number of readers of an object

Example: Immutable Borrow

```
fn a() {
let x = Foo::new();   // x is the owner
let y = &x;              // y is an immutable borrow of x;  x is still the owner.
bar(y) ;       // pass an immutable borrow to bar; the borrow's lifetime is the lifetime of bar
}


fn bar(z: &Foo) {
  ... =  .. z ...                    // we can read from z in bar as many times as we like
  //     global.f = z        storing z somewhere that outlives bar will generate a type error
}
```

A Problem

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

This Rust function returns the longer of two strings

As written, the function does not type check!

Why?

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

What is the lifetime of the result?

It is either the lifetime of x or the lifetime of y

How can this lifetime information be represented?

Digression: Type Checking If-Then-Else

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

$$\frac{A \vdash e_1 : \text{Bool} \qquad A \vdash e_2 : T \qquad A \vdash e_3 : T}{A \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T}$$

If-Then-Else requires the types of the two branches to be the same

Analogously, an ownership type system requires the lifetimes of the two branches to be the same

Lifetime Annotations

```
fn longest<‘a>(x: &’a str, y: &’a str) -> &’a str
{
   if x.len() > y.len() {
      x
   } else {
      y
   }
}
```

- The function is templated on a *lifetime annotation*
- Requires that the two arguments have the same lifetime
  - And thus the result has that lifetime, too
- This version type checks

Discussion

- Ownership rules are very restrictive
  - Program must be *linear* in owned objects
  - Exactly one owner at all times

- Three techniques help in writing legal and useful programs:
  - Using immutable data wherever possible
  - Deep copies are OK (*cloning*)
  - Borrowing creates a reference that can be used
    - Does not transfer ownership
    - Implies a borrowed reference cannot deallocate an object
    - The owner cannot deallocate an object until all borrowed references are returned
    - Borrowed references have a different syntax and type

Ownership in Practice

- Ownership has been studied for > 20 years
- Rust is the first full language to support ownership types
  - The major new feature
- Experience is that Rust's ownership system helps
  - Enables manually managed memory without the bugs
  - Makes it possible to write efficient and correct code
  - Ownership types are the key
  - Which is not to say ownership is always easy to use
    - Programmers need to reason about lifetimes
    - Rust's type inference helps a lot
    - But sometimes lifetimes are not inferred and explicit lifetime annotations are needed

Coda: Interfaces

Review: Single Inheritance

```
Class Foo {
    method f(a: WhatsIt, b: WhoseIt) { … some code … }
}

Class Bar inherits Foo {

}

x: Whatsit;
y: Whoseit;
(new Bar).f(x,y)          // Bar also provides f, inherited from Foo
```

Review: Single Inheritance w/Override

```
Class Foo {
    method f(a: WhatsIt, b: WhoseIt) { … some code …}
}

Class Bar inherits Foo {
    method f(a: WhatsIt, b: WhoseIt) {… some completely different code … }
}

x: Whatsit;
y: Whoseit;
(new Bar).f(x,y)        // Bar provides an f different from Foo's f, but with the same interface
```

Abstract Methods

```
Class Foo {
    virtual method f(a: WhatsIt, b: WhoseIt);    // no code --- only the interface is declared
}

Class Bar inherits Foo {
    method f(a: WhatsIt, b: WhoseIt) {… some code implementing the interface … }
}

Class Bazz inherits Foo { ... another class implementing Foo’s interface in a different way … }

x: Whatsit;
y: Whoseit;
(new Bar).f(x,y)
```

The Evolution from Inheritance to Interfaces

- Single inheritance was discovered to be quite limiting
  - Only can inherit from one parent class
  - But many types would naturally inherit from multiple classes
    - A University is both a NonProfit and a School

- Completely abstract classes became popular
  - All methods are abstract
  - Separate declaration of the interface from all implementations

- Recently object systems have moved to
  - Declare interfaces, a named set of abstract methods
  - Types can implement any number of (previously declared) interfaces
    - E.g., University implements NonProfit, School { … }

Rust Traits

- Traits are the way to do inheritance of functionality in Rust
  - Traits declare abstract interfaces
  - Types implement these interfaces

- Inspired by Haskell type classes
  - And similar to Java interfaces

Traits Example (from ``Rust By Example’’)

```
struct Sheep { naked: bool, name: &'static str }
trait Animal {
 // Traits declare types of methods any implementor type must provide
// Associated function signature; `Self` refers to the implementor type.
   fn new(name: &'static str) -> Self;
   fn name(&self) -> &'static str;
   fn noise(&self) -> &'static str;
   // Traits can provide default method definitions.
   fn talk(&self) {
      println!("{} says {}", self.name(), self.noise()); }
}
impl Sheep {
   fn is_naked(&self) -> bool { self.naked }
   fn shear(&mut self) {
      if self.is_naked() {
         println!("{} is already naked...", self.name()); } else {
         println!("{} gets a haircut!", self.name);
         self.naked = true;
    }}}
```

```
// An implementation must explicitly declare what trait it is implementing
impl Animal for Sheep {
   // `Self` is the implementor type: `Sheep`.
   fn new(name: &'static str) -> Sheep {
       Sheep { name: name, naked: false }
   }
   fn name(&self) -> &'static str { self.name }
   fn noise(&self) -> &'static str {
      if self.is_naked() { "baaaaah?” } else { "baaaaah!” }
   }

// Override default method.
   fn talk(&self) { println!("{} pauses briefly... {}", self.name, self.noise()); }
}
```

Summary

- Rust provides static memory management
  - Memory safety with the efficiency of C/C++ code
  - Key is reasoning about different classes of pointers (owners/borrows) and their lifetimes

- And a modern interface system
  - Traits allow declaration/implementation of flexible class-like interfaces

- Rapidly gaining ground in industry
  - There are millions of Rust programmers today