CMSC 330: Organization of Programming Languages

Traits
Overview

• **Traits** abstract behavior that types can have in common
  – Traits are a bit like *Java interfaces*
  – But we can implement traits over any type, anywhere in the code, not only at the point we define the type

• **Trait bounds** can be used to specify when a generic type must implement a trait
  – Trait bounds are like *Java’s bounded type parameters*
Defining a Trait

• Here is a trait with a single function

```rust
pub trait Summarizable {
    fn summary(&self) -> String;
}
```

• Specify `&self` for “instance” methods
  – Can also specify “associated” methods
    » Like `static` methods in Java

• Equivalent in Java:

```java
public interface Summarizable {
    public String summary();
}
```

Note: The keyword `pub` makes any module, function, or data structure accessible from inside of external modules. The `pub` keyword may also be used in a `use` declaration to re-export an identifier from a namespace.

Note that we make the entire trait public, not individual elements of it.
Implementing a Trait on a Type

- **name of trait**
  - `impl Summarizable for (i32,i32)`
- **type on which we are implementing it**
  - `(i32,i32)`
- **trait method invocation**
  - `fn foo() { let y = (1,2).summary(); //"3" let z = (1,2,3).summary(); //fails }`
- **trait method body**
  - `fn summary(&self) -> String {
      let &(x,y) = self;
      format!("{}","{},{})",x+y)
  `
Default Implementations

• Here is a trait with a default implementation

```rust
pub trait Summarizable {
    fn summary(&self) -> String {
        String::from("none")
    }
}
impl Summarizable for (i32, i32, i32) {}
```

```rust
fn foo() {
    let y = (1, 2).summary(); // "3"
    let z = (1, 2, 3).summary(); // "none"
}
```
Trait Bounds

• With generics, you can specify that a type variable must implement a trait

```rust
pub fn notify<T: Summarizable>(item: T) {
    println!("Breaking news! {}",
        item.summary());
}
```

– This method works on any type \( T \) that implements the `Summarizable` trait

• This is a kind of subtyping: \( T \) can have many methods but at the least it should implement those in the `Summarizable` trait
Trait Bounds: Like Java Bounded Parameters

• Equivalent in Java

```java
<T extends Summarizable>
void notify(T item) {
    System.out.println("Breaking news! "+
    item.summary());
}
```

- This generic method works on any type T that implements the `Summarizable` interface (which we showed before)

```java
public interface Summarizable {
    public String summary();
}
```
Generics, Multiple Bounds

• Trait implementations can be generic too
  
  ```rust
code
  pub trait Queue<T> {
    fn enqueue(&mut self, ele: T) -> (); ...
  }
  impl<T> Queue<T> for Vec<T> {
    fn enqueue(&mut self, ele:T) -> () {...} ...
  }
  ```

• Generic method implementations of structs and enums can include trait bounds

• Can specify multiple Trait Bounds using +
  
  ```rust
  fn foo<T:Clone + Summarizable>(...) -> i32 {...} or
  fn foo<T>(...) -> i32 where T:Clone + Summarizable {...}
  ```
(Non)Standard Traits

- We have seen several standard traits already
  - **Clone** holds if the object has a `clone()` method
  - **Copy** holds if assignment duplicates the object
    - I.e., no ownership transfer, as with primitive types
  - **Move** holds if assignment moves ownership
    - I.e., because assignment doesn’t copy it all; the default
  - **Deref** holds if you can dereference it
    - I.e., it’s a primitive reference, or has a `deref()` method

- There are other useful ones too
  - **Display** if it can be converted to a string
  - **PartialOrd** if it implements a comparison operator

*Note*: Several of these traits indicate special treatment by the compiler, e.g., **Move** and **Copy**; they go beyond the indication that an object implements particular methods.
Putting all Together

• Finds the largest element in an array slice
  – Generic in the type \( T \) of the contents of the array

```rust
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    let mut largest = list[0];
    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }
    largest
}
```

Requires `PartialOrd` trait

Requires `Copy` trait to not transfer ownership
Putting all Together

• Finds the largest element in an array slice
  – Generic in the type $T$ of the contents of the array

```rust
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    ...
}
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];
    let result = largest(&number_list);
    println!("The largest number is {}", result);
    let char_list = vec![\'y\', \'m\', \'a\', \'q\'];
    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

prints

The largest number is 100
The largest char is y
trait Trait {
    fn p(&self);
}

impl Trait for u32 {
    fn p(&self) { print!("1"); }
}

let x = 100; // inferred as u32
x.p();

A. 100
B. 1
C. Error
trait Trait {
    fn p(&self);
}

impl Trait for u32 {
    fn p(&self) { print!("1"); }
}

let x=100; // inferred as u32
x.p();

A. 100
B. 1
C. Error