Rust Closures

CPSC 310 - Programming Languages

Closures

- Rust has local functions and closures

```
fn moveit(l: bool, x: i32) -> i32 {
    let left = |x| x - 1;
    fn right(x: i32) -> i32 { x+1 };
    if l { left(x) }
    else { right(x) }
}
```

- OCaml equivalent

```
let moveit l x =
  let left = fun x -> x - 1 in
  let right = fun x -> x + 1 in
  if l then left x
  else right x
```

Limits of Type Inference

- Rust infers non-polymorphic types

```
let id = |x| x;
let x = id(1); // infers x: i32
let y = id("hi") // fails: &str != i32
```

- OCaml infers polymorphic types

```
let f = fun x -> x in  (* 'a -> 'a *)
let x = id 1 in
let y = id "hi" in
...
```

Iteration with the `Iterator` Trait

- Example

```
let a = vec![1, 2, 3, 4, 5];
for e in a.iter() {
    println!("value: {}", e);
}
```

- The `iter()` method returns an iterator, that is, a value with the `Iterator` trait

```
trait Iterator {
    type Item; // this is an associated type
    fn next(&mut self) -> Option<Self::Item>;
    ... // default method impls
}
```

The `for` Syntax

- Each call to `next` advances the iterator, so it has to be `mut`

```
let a = vec![1, 2]
let mut iter = a.iter();
assert_eq!(iter.next(), Some(&1));
assert_eq!(iter.next(), Some(&2));
assert_eq!(iter.next(), None);
```

- Calls to `next` produce immutable references to the values in `a`
  - can call `into_iter` or `iter_mut` on `a` to get different kinds of references

`Iterator` Adaptors

- We can make one iterator from another
  - An iterator is consumed as it is used; it is lazy
- This is a pattern for higher-order programming
  - `i.map(f)` produces an iterator returning `f(e)` for each of `i`'s elements `e`
  - `i.filter(f)` produces an iterator for `i`'s elements `e` such that `f(e) == true`
  - `i.collect` converts an iterator into a vector
  - `i.fold(a, b)` is like OCaml's `fold_right`
  - `zip`, `sum`, etc.

Examples

```
let a = vec![10, 20];
let i = a.iter();
let j = i.map(|x| x+1).collect(); // [11,21]
let k = a.iter().fold(0, |a, x| x - a); // 10
for e in a.iter().filter(|&&x| x == 10) {
    println!("{}", e);
} // prints 10
```

Iterator Notes

- We can make our own iterators
  - Implement the `Iterator` trait
  - Several examples in the Rust Book
- Iterators perform extremely well
  - Better than for loops with explicit indices
  - The compiler optimizes the code it generates
  - So, feel free to program using map, fold, zip, etc.

Closures: Passing as Arguments

- Each closure has a distinct type
  - Even if two closures have the same signature, their types are considered different
- To specify the type of a closure (for a function parameter, say), use generic traits with bounds: `Fn t`, `FnMut t`, or `FnOnce t`
- Functions implement the above trait bounds too

The Fn Trait

- Example

```
fn app_int<T>(f: T, x: i32) -> i32
    where T: Fn(i32) -> i32
{ f(x) }

fn main() {
    println!("{}", app_int((|x| x-1), 1));
}
```

- Cannot write

```
fn app_int(f: (i32) -> i32, x: i32) -> i32
{ f(x) }
```

Using the `Fn` Trait Polymorphically

- Example

```
fn app<T, U, W>(f: T, x: U) -> W
    where T: Fn(U) -> W
{ f(x) }

fn main() {
  println!("{}", app((|x| x-1), 1)); // i32
  let s = String::from("hi ");
  println!("{}", app((|x| x + "there"), s)); // String
}
```

Capturing Free Variables

- Example

```
fn main() {
    let x = 4;
    // closure captures x
    let equal_to_x = |z| z == x;
    let y = 4;
    assert!(equal_to_x(y)) // true
}
```

- Note: fails if `equal_to_x` is defined as a local functions; local functions do not have an environment
- Complication
  - Capturing it could move it or borrow it
  - Use various `FnX` traits to specify what to do

Distinguishing `Fn` Trait Bounds

- `FnOnce t` (where `t` is a func type)
  - Consumes the variables it captures from its enclosing scope, that is, moves or copies them
  - Thus, can only be called once
- `FnMut t`
  - Borrows captured variables mutably
- `Fn t`
  - Borrows captured variables immutably, or copies
  - Try this bound first, follow the compiler's advice if it does not work

FnOnce Example

- Example

```
let x = String::from("hi");
let add_x = |z| x + z; // captures x; is FnOnce
println!("x = {}", x); // fails
let s = add_x(" there"); // consumes closure
let t = add_x(" Bob"); // fails, add_x consumed
```