Concepts in Programming Languages

Additional non-examinable notes

Alan Mycroft

Computer Laboratory
University of Cambridge

CST Paper 7: 2021–2022 (Easter Term)

www.cl.cam.ac.uk/teaching/2122/ConceptsPL/

C++ lambdas: variable capture (value or reference)

```
// LLVM use: c++ --std=c++14 to get lambda support

#include <iostream>
int main()
{   int a=0,b=0;
    // C++ use of `[]' (lambda)needs to know how
    //                 free variables are bound:
    auto f = [a,&b](int x) ->int { return x+a+b;};
    a++; b+=10;
    std::cout << "f(42)=" << f(42) << std::endl;
    // gives "f(42)=52" -- think why...
    return 0;
}
```

Notes:

- `auto f = [](int x) ->int { return x+a+b;};` gives “error: variable ’a’ cannot be implicitly captured in a lambda with no capture-default specified.”
- The type of f is a C++ ‘functor’, but that’s another story.

Topic XII

Concepts growing in importance in 2022

- Type-managed storage [example: Rust]
- Resumable exceptions/algebraic effect handlers [examples: Eff and Koka]

Approaches to storage allocation

- *Manual*, e.g. C/C++
  Danger: user-incompetence (use after free)
- *Automatic*, e.g. Java, Python
  Danger: unexpected delays (GC in a flight controller???)
- *Ban it*, MISRA (motor industry embedded coding standard)
  Rule 18-4-1 “Dynamic heap memory allocation shall not be used.”
- *Type-managed* – Rust
  Upside: type system ensures memory- and thread-safety and automatically adds calls to deallocate storage. No GC.
  Downside: search online for “Rust hard to learn” – but this is perhaps an advantage for smart programmers!
  Fact: Rust is ranked 19 in the 2022 Redmonk language rankings – so there are plenty of smart programmers and interesting companies about!

Storage management is more than allocation and deallocation

Regardless of manual or automatic deallocation there's a wider issue: *ownership* (related to the sweet spots on slides 222)

- If I pass a mutable object to be incorporated into a global datastructure, then logically the call transfers ownership from the caller to the callee, so I should never refer to it again – just like `free`?
- When an API call returns me a record (which I plan to mutate) is the caller or returner (callee) responsible for copying it?
- Can we check this sort of property at compile time? Types?

Reasoning about ownership is more-general than reasoning about manual deallocation (freeing an object is just like passing ownership to the pool of free memory).

Type Systems – weakness

In traditional type systems $\Gamma \vdash e : t$, variables have the same type throughout the scope that introduce them.

This means that the three errors in the following program can't be detected by the type system:

```
{ char *x = malloc(10); // x has type char *
  foo(x);         // x still has type char *
  free(x);        // x has type char * (but should not??)
  foo(x);         // a use-after-free disaster...
  x = malloc(20); // type char * is right again for x
  x = malloc(30); // an unfaulted memory leak
}                 // and another one (x gone out-of-scope)
```

Replacing `free(x)` with `AddToGlobalDataStructure(x)` shows classical types are equally weak at controlling sharing of data in languages like Java (details on next slide).

Type Systems – weakness (2)

```
{ char *x = malloc(sizeof(SomeRecord));
  x->field1 = 4; x->field2 = 5;
  AddToGlobalDataStructure(x);
  // x = malloc(sizeof(SomeRecord));
  x->field1 = 8; x->field2 = 9;
  AddToGlobalDataStructure(x);
  // free(x);
}
```

Is there a bug in this code? [Almost certainly.] But what is it?

- AddToGlobalDataStructure wants to take ownership of `x`, so the problem is due to commenting out line 4?
- AddToGlobalDataStructure merely reads the fields of `x`, so the problem is due to commenting out line 7?
- How can we document this formally? [Answer: "Use Rust"]

Solutions

- Code it all in the type system: *linear type*, *substructural types*, *separation logic*.
- Treat reference types as having two attributes: *type* and *ownership*.
- Just different phrasing, Rust follows the latter.
- Rust performs type-checking then runs the *borrow checker*.

Borrow? Well, if we have owners we might also lend and borrow, right?

[Thanks to Brendan Coll (CST Part II 2021/22) for his comments on these notes on Rust.]

Rust by example

(from the manual)

- Rust types look a bit like Java types with C-like qualifiers, but be careful.
- `Box<i32>` is like `ref` in ML or a boxed `int` in Java:

```
fn create_box() {
  let _box1 = Box::new(3i32);   // ref to heap int
  // ’_box1’ is destroyed (’dropped’) and its memory freed
  // Resembles C++ RAII/destructors.
}
// destroy_box takes ownership of an item of
// heap-allocated memory (default call-by-value)
fn destroy_box(c: Box<i32>) {
  println!("Destroying a box that contains {}", c);
  // ’c’ goes out of scope here and is deallocated.
}
```

- Passing an object by value passes its ownership, and owners can destroy things (think cars etc.)

But but but . . .

Copying and passing by value as similar

Rust calls `let b=a` below (and passing by value) a *move*.

```
fn main() {
  let x = 5u32; // stack allocated int
  let y = x;    // copy x to y
  println!("x is {}, and y is {}", x, y); // use both
// BUT:
  let a = Box::new(5i32); // stack ref to heap-allocated int
  println!("a contains: {}", a); // (borrows the ref)

  let b = a; // copy a to b -- transfers ownership
  println!("a contains: {}", a); // error, ’a’ not owner

  destroy_box(b);
  println!("b contains: {}", b); // error, ’b’ not owner
}
```

Can we really write programs?

Mutability

Mutability of data can be changed when ownership is transferred (think why). Note `*` is not quite like C:

```
fn main() {
  let immutable_box = Box::new(5u32);
  println!("immutable_box contains {}", immutable_box);

  *immutable_box = 4; // error

  // *Move* the box, changing the ownership (and mutability)
  let mut mutable_box = immutable_box;

  println!("mutable_box contains {}", mutable_box);  // 5
  *mutable_box = 4;
  println!("mutable_box now contains {}", mutable_box); // 4
}
```

Borrowing

```
fn borrow_box(x: &Box<i32>) {
    println!("A borrowed box (see below): {}={}" &x, x); }

fn borrow_twice(x: &Box<i32>, y: &Box<i32>) {
    println!("Borrow two boxes: {}, {}", x, y); }

fn borrow_and_eat(x: &Box<i32>, y: Box<i32>) {
    println!("Borrow box {}, and destroy box {}", x, y); }

fn main() {
    let b = Box::new(5i32);
    let c = Box::new(6i32);
    borrow_box(&b); // 5=5
    borrow_twice(&b,&c); // 5,6
    borrow_twice(&b,&b); // 5,5
    borrow_and_eat(&b,c); // 5,6
    borrow_and_eat(&b,b); // error (borrow checker)
}
```

Subtlety: `println!` is a macro which implicitly inserts `&` before borrowed arguments, see `borrow_box` above using `x` twice.

Borrowing – more details

- Can borrow (`&mut`) a mutable object once, or an immutable object many times (cf. Multiple Reader Single Writer (MRSW) in concurrent systems). Good for concurrent access, and also for the AddToGlobalDataStructure example earlier.
- Rust's borrowing discipline prevents unsafe uses of aliasing.
- Can borrow parts of an object
- All borrowing must be completed before ownership can be transferred
- Gives memory safety – no referencing freed memory, and pretty well avoids memory leaks. [Subtleties: you can break memory safety by using `unsafe` to cheat the type system; you can leak memory if you really try but not using examples we have seen.]
- Rust advocates claim that these rules are an acceptable the sweet spot to ensure memory safety.

Resumable exceptions

Normally called (Algebraic) Effect Handlers

Start by revisiting exceptions (SML, OCaml, Java are all semantically similar):

- `exception Foo;` declare exception `Foo`
- `raise e;` raise an exception
- `try e catch Foo => e';` handle exception `Foo`

Exceptions behave syntactically like constructors (for type `exn` in SML/OCaml and subclasses of `Throwable` in Java), and hence may take parameters; the `catch` part of `try` is like pattern matching.

[Thanks to Dan Gooding (CST Part II 2021/22) for examples and general discussion; see also his Part II project on Koka.]

Exceptions: dynamic or static scoping?

- A mixture!
- Declaring an exception is statically scoped

  ```
  exception Foo;
  ```

- Handling an exception is like dynamic scoping

  ```
  exception Foo;
  fun f():int = raise Foo;
  fun g() = (try f() catch Foo => 1) + f();
  fun main() = (try g() catch Foo => 42)
  ```

  gives 42.

Subtlety: OCaml doesn't syntactically allow local exception declarations, but wrapping the exception in a `module` circumvents this.

Resumable exceptions

- Resumable exceptions are generally called `effect`s. Why? Can see calls to side-effecting operations like IO as having exceptional behaviour handled by OS (a system call), and then your program is resumed.
- In general effects have result types to allow resume-with-a-value (think `read()`).
- We now look at Koka programs using resumable exceptions to model `yield`, *dynamic scoping* and Prolog non-determinism.
- We use `resume` to return a value from an effect
- Koka subtlety: effects can be declared as `ctl` or `fun`. Declaring an effect as `fun` is syntactic sugar – such code desugars to use `ctl` and inserts `resume` automatically – but also allows the compiler to generate more efficient code.

A simple Koka program

```
// A generator effect with one 'fun' operation
effect yieldeff
  fun yield( x : int ) : ()

// Traverse a list and yield the elements
fun traverse( xs : list<int> ) : yieldeff ()
  match xs
    Cons(x,xx) ->  yield(x); traverse(xx)
    Nil        -> ()

fun main() : console ()
  with fun yield(i : int)
    println("yielded " ++ i.show)
  [1,2,3].traverse
```

[Modified from https://koka-lang.github.io/koka/doc/index.html]
Gives "yielded 1" "yielded 2" "yielded 3"

Using `ctl` exceptions

An optionally resumable (`ctl`) effect

```
effect yieldeff
  ctl yield( x : int ) : ()

// Traverse a list and yield the elements
fun traverse( xs : list<int> ) : yieldeff ()
  match xs
    Cons(x,xx) ->  yield(x); traverse(xx)
    Nil        -> ()

fun main() : console ()
  with ctl yield(i : int)
    if (i>2) then () // don't resume
    else  // unusual syntax to reflect fun desugaring:
      resume(println("yielded " ++ i.show))
  [1,2,3,4].traverse
```

Gives "yielded 1" "yielded 2"

Effect names vs. effect-operation names

Minor naming subtlety

- Why did I distinguish `yield` from `yieldeff`?
- Pedagogy! Just like avoiding `list : int list` when learning ML
- Experts tend not to bother when an effect only has a single effect operation (like `yield`)
- But necessary for effects with multiple operations:

```
effect state<a> {
  ctl get() : a
  ctl set(s : a) : ()
}
```

Dynamic scoping – using `ctl` effects

```
// Simulation of dynamic scoping
effect dyneff
  ctl dynvar (s : string) : int

fun f1() : dyneff int
  dynvar("x") + dynvar("y") + dynvar("z");

fun f2(x : int) : dyneff int
  with ctl dynvar(s)
    if s=="x" then resume(x)  // x visible as dynamic
    else if s=="z" then resume(20) // bind z to 20
    else resume(dynvar(s)); // look in outer scope
  f1()

fun foo() : dyneff int
    dynvar("x") + f2(500) + 1

fun main() : console ()
  with ctl dynvar(s)
    resume(1000)   // unbound vars give 1000
  println("foo gave " ++ foo().show) // 2521
```

Dynamic scoping – using `fun` effects

```
// Simulation of dynamic scoping using fun effects
effect dyneff
  fun dynvar (s : string) : int

fun f1() : dyneff int
  dynvar("x") + dynvar("y") + dynvar("z");

fun f2(x : int) : dyneff int
  with fun dynvar(s)
    if s=="x" then x else if s=="z" then 20
    else dynvar(s);  // look in outer scope
  f1()

fun foo() : dyneff int
    dynvar("x") + f2(500) + 1

fun main() : console ()
  with fun dynvar(s)
    1000            // unbound vars give 1000
  println("foo gave " ++ foo().show) // 2521
```

Prolog style backtracking as effects

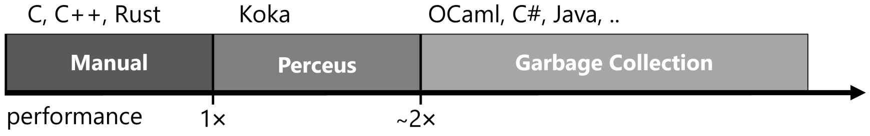
```
fun myor(a: bool, b: bool) a&&b;   // define non-short-circuit OR

effect choose
  ctl flip() : bool  // new: handlers resume flip() more than once

fun mystery() : <choose,console> bool // a function to test
  val x = flip()
  val y = flip()
  val z = flip()
  val myst = !x && y && !z
  // this debug line causes the uses of ’console’
  println(x.show ++ y.show ++ z.show ++ "->" ++ myst.show)
  myst

fun satisfiable(p : () -> <choose,console> bool) : <console> bool
  // Try all inputs to see if p satisfiable;
  // the order is: xyz = 000, 001, 010, 011, 100, ...
  with ctl flip()
    // for each input variable, try both values:
    myor(resume(False), resume(True)) // short-circuit OR differs(!)
  p()

fun main()
  satisfiable(mystery).println // True
```

The bigger picture

- We’ve focused on effects including exceptions (never resume), fun effects (resume once), and Prolog-style multiple resumptions.
- Effects are a structured use of continuations.
- Koka has a *type system which models possible effects* (Haskell notion of ‘pure’ includes effects `{div, exn}`):

fun sqr :	(int) -> total int	// total: mathematical total function
fun divide :	(int,int) -> exn int	// exn: may raise an exception (partial)
fun turing :	(tape) -> div int	// div: may not terminate (diverge)
fun print :	(string) -> console ()	// console: may write to the console
fun rand :	() -> ndet int	// ndet: non-deterministic

- Various other goodies: type and effect polymorphism; and Perceus compiler store re-use optimiser:



C, C++, Rust
Koka
OCaml, C#, Java, ..
Manual
Perceus
Garbage Collection
performance
1×
~2×

Places to look for more detail

- `https://www.rust-lang.org/`
- `https://www.eff-lang.org/`
- `https://koka-lang.github.io/`

Such languages (or subsets of their features) can make interesting Part II projects.