Comp 411
Principles of Programming Languages
Lecture 30
Retrospective On Program Design

Corky Cartwright

April 13, 2022

Common Issues

- Python is pragmatic rather than a principled basis for teaching program design. The biggest weakness of Python is that it does not obey lexical scoping rules or any coherent alternative. Even dynamic scoping (a broken variant of lexical scoping introduced to simplify the implementation of passing functions as arguments) has an easily described if ugly semantics. Python is the only mainstream high-level language since Fortran with that does not use either lexical scoping or its crippled sibling, dynamic scoping.
- C, C++, and Java all obey lexical scoping rules but restrict lexical scope by limiting procedure and method definitions to the top-level (except via the complex backdoor of inner classes in the case of Java and recent versions of C++).
- In Java and C++, the absence of nested methods (which can be achieved at the cost of additional semantic complexity by using inner classes) is typically addressed by passing objects as parameters. In designing OO programs, it is easy to bundle the requisite values in a small number of objects that serve as the receiver and parameters in method calls.

Common Issues

- A good case can be made for defining a Java or C++ class as an inner class when class instances nced access to fields or method parameters in the enclosing class. (So-called static inner classes should not be called “inner classes” because the are simply top-level classes with names that are qualified by the name of the enclosing class.)
- Anonymous inner classes vs. method delegates (Java vs C#). C# putatively improved on Java since it was designed after Java as a Java-killer, but I prefer inner classes to method delegates as do many OO developers that I know.
- Subtyping among (parameterized) generic types. In structurally typed functional languages (e.g., the ML family of languages), subtyping is prohibited. In OO languages, it is surprisingly subtle because subtyping (via inhertiance) abounds yet function types have the following subtyping property:

  $A' \rightarrow B'$ is a subtype of $A \rightarrow B$ iff

  A' is a *supertype* of A, and B' is a subtype of B

  In the context of mutable cells, no subtyping is possible if cells are both read and written. Contract preservation in derived classes includes preserving the types of inherited methods and fields.

Java vs Algol Runtimes

- The Java runtime is simpler than the classic Algol 60 runtime which was designed to support lexical scoping in nearly full generality.
- The missing generality in the Algol 60 run-time is the lack of support for procedures/functions as “first-class” values that can be bound to variables, returned as results of procedures/methods, assigned to fields of objects/structures/records.
- Guy Steele completely solved this problem by
  - Assuming the presence of a heap (for dynamic allocation of objects/structures/records)
  - Using lightweight closures that copy the invariant bindings [values and immutable cell addresses] corresponding to the specific free variables in the procedure/method code instead linking to th the entire closing environment)
  - Representing the invariant binding of each closed over mutable variable in an activation record by using a level of indirection (placing the actual mutable cell in the heap) so it is never deallocated as long as it is accessible.
  - Guy Steele’s generalized Algol 60 runtime is now the standard runtime for languages that support nested procedures and functions (like Swift).
- Since methods cannot be nested in Java (except via inner classes), there is no static link in the activation record for a method invocation. C and C++ share this “simplification”. (Well-written code is not any simpler!)

Copying Collectors III

Pseudo-code for a Cheney collector:

Flip identity of new and old space.

Copy the target objects of roots to new space, updating the root pointer cells, creating forwarding pointers in old copies of these objects, **`front`** and **`rear`** pointers to the sequentially allocated queue formed by the copied objects in new space, and a **`next`** pointer to the first word following object(rear).

```
while (front <= rear) {
  for each pointer field p in object(front) {
    if (object(p) has a forwarding pointer p')
      update field p to p';
    else {
      copy object(p) to next;
      update field p to next;
      rear := next;
      advance next;
    }
  advance front;
}
```

Copying Collectors IV

- Allocation of a new object in new space: do the following atomically
  - Save old value of next;
  - Initialize the allocated object, incrementing next by size of allocated object;
  - Return the old (saved) value of next as the address of the allocated object;
  - Pros and cons of Cheney collection:
  - GC is efficient: runs in time proportional to amount of live data (size of new heap). Amount of work for each byte of copied data is small.
- Half of available heap space is lost!
- Can we combine advantages of Cheney collection and mark-and-compact?
- Yes! Use generational "scavenging" (collection) where Cheney collection is used to manage a nursery (a small contiguous subheap) where new objects area allocated. Since Cheney collection is used to manage nursery storage, half of the nursery is unused but the loss is small because the nursery is small. After an object lives for specified number of nursery collections, it is *promoted* to the main heap, which is managed using some form of mark-and-compact collection. Tbe details of generational collection include a few subtleties discussed on subsequent slides.