Adding Lifetime Checking to Chapel

Michael Ferguson, Cray Inc (presenting author)

One of the defining features of the Rust language is its support for compile-time checking to guarantee memory safety. This approach relies on a component of the Rust compiler called the *borrow checker* [1]. Other languages have developed similar compile-time checking strategies of their own – for example D scoped pointers [2] and the C++ effort related to the Core C++ Guidelines that adds static checking for pointer lifetimes [3].

This compile-time approach is interesting because it presents an alternative to garbage collection as a means of preventing use-after-free errors. Use-after-free errors are common cause of memory corruption and these types of errors can be difficult to debug. Anyway, one of the benefits of a garbage collection is that use-after-free errors simply are not possible, since any reference to an object will prevent its memory from being reclaimed. In this way, garbage collection offers a productivity benefit beyond simply allowing programmers not to worry about where to put a 'free' call.

Chapel is not a garbage collected language – and in fact adding a garbage collection might run counter to its performance goals. Distributed garbage collection is a challenging topic and it may not be possible to create a distributed garbage collection system with low enough overhead to be suitable for high performance computing.

Since Chapel doesn't have garbage collection, Chapel class instances are manually managed (i.e. the Chapel programmer puts a call to 'delete' in order to free their memory). Recent work has added Owned and Shared records to simplify the task of calling 'delete' – these records rely on record destruction to call 'delete' at an appropriate time [4]. However, these records do nothing to prevent use-after-free errors. Can the Chapel compiler be improved to detect use-after-free errors?

This talk will present new work towards improving the Chapel language and compiler to include detection of use-after-free errors at compile-time. Language changes will include adding new variations of class types, changing the type returned by 'new', and adding syntax to specify the lifetime of a value returned by a function. While new syntax allows the exact specification of these properties, it's critical for usability of these features that the language and compiler offer reasonable defaults when the explicit syntax is not used. One area of particular interest is the rules the compiler uses to infer the lifetime of the value returned from a function when it is not specified directly.

[1] https://doc.rust-lang.org/book/references-and-borrowing.html
[2] https://github.com/dlang/DIPs/blob/master/DIPs/DIP1000.md
[3] http://www.stroustrup.com/resource-model.pdf
[4] https://chapel-lang.org/docs/latest/modules/packages/OwnedObject.html