Adding Lifetime Checking to Chapel

Michael Ferguson, Cray Inc.

CHIUW 2018

May 25, 2018

Safe Harbor Statement

This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.

Purpose of this effort

Memory Management Strategies Scorecard

Garbage Collection	'delete'
+ safety guarantees + eliminates memory leaks + eliminates double-delete + eliminates use-after-free	– more errors possible – failure to delete results in leaks – double-delete possible – use-after-free possible
+ ease-of-use + no need to write 'delete'	– more burden on programmer – think about 'delete'
– implementation challenges due to distributed memory & parallelism	+ simpler implementation
– performance challenges – stop-the-world interrupts program – concurrent collectors add overhead – scalability may prove difficult	+ predictable, scalable performance

- **Based on these tradeoffs, Chapel started with 'delete'**

First Step: Owned and Shared

- **General-purpose wrapper records help avoid ‘delete’**
- **Owned: uses a single-owner pattern to manage lifetime**
  - contained class instance deleted when ‘Owned’ goes out of scope
  - assignment and copy initialization are destructive ownership transfers
- **Shared: uses reference-counting to manage lifetime**
  - contained class instance deleted when all ‘Shared’ copies destroyed
  - assignment and copy initialization share ownership
- **Introduced in version 1.15**

Owned and Shared Scorecard

- **Owned and Shared remove the need to write 'delete' but do not address memory safety**

Owned and Shared

– not much safer than 'delete'
  – double-delete possible
  – use-after-free possible

+ no need to write 'delete' – have to mark variables/fields as Owned/Shared
+ manageable implementation
+ low impact on execution-time program performance

Background: Rust

- **Rust's approach prevents memory errors at compile time**
  - programs that might have a use-after-free result in compilation error
  - its *borrow checker* is the component raising these errors
- **Rust's approach also prevents race conditions**
  - since race conditions can introduce memory errors
- **Rust programmers can also write *unsafe* code**
  - provides a way to opt out of the above checking
  - expectation is that unsafe code is carefully inspected

Motivating Question

- **Can Chapel include something Rust-like?**
  - compile-time detection of use-after-free?
- **The Big Issue: Complete Checking and Race Conditions**
  - recall that a race condition can introduce a use-after-free error
  - For example:

```
proc test() {
  var myOwned = new Owned(new MyClass());
  var b = myOwned.borrow();
  cobegin with (ref myOwned) {
    { myOwned.clear(); }     // deletes instance
    { writeln(b); }          // races to use instance before delete
  }
}
```

Complete Checking and Race Conditions

- **Should Chapel rule out race conditions at compile time?**
- **A worthy goal, but the Rust strategy doesn't fit Chapel**
  - only one mutable reference to an object can exist at a time
  - if a mutable reference exists, no const references to that object
- **Such a strategy in Chapel would make these illegal:**

```
forall a in A { a = 1; }
forall i in 1..n { A[i] = i; }
forall i in 1..n { B[permutation(i)] = A[i]; }
```

- **Could a different strategy detect these race conditions?**
  - Maybe, but it would be difficult
  - Can the compiler prove that 'permutation' is a permutation?
  - If not, how would that be communicated to the compiler?

General Goal

General Goal

- **Add incomplete compile-time checking to gain some of the benefits of garbage collection**

Proposal: Lifetime Checking
+ helps with safety + eliminates many memory leaks + eliminates many double-delete + eliminates many use-after-free – but doesn't catch all cases
+ no need to write 'delete' – have to mark variables/fields as owned/shared/borrowed
+ manageable implementation
+ low impact on execution-time program performance

– have to mark variables/fields as
owned/shared/borrowed

New Keywords

new keyword	meaning
unmanaged	the instance in manually managed and needs to be 'delete'd at some point
owned	the instance is deleted at end of scope
shared	the instance is reference counted
borrowed	the instance is managed elsewhere and this reference does not impact its lifetime

New Keywords

Class types meant unmanaged prior to this work

new keyword	meaning
unmanaged	the instance in manually managed and needs to be 'delete'd at some point
owned	the instance is deleted at end of scope
shared	the instance is reference counted
borrowed	the instance is managed elsewhere and this reference does not impact its lifetime

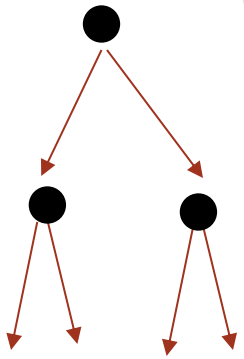
New Keywords

new keyword	meaning
unmanaged	the instance in manually managed and needs to be 'delete'd at some point
owned	the instance is deleted at end of scope
shared	the instance is reference counted
borrowed	the instance is managed elsewhere and this reference does not impact its lifetime

Class types default to borrowed after this work

Binary Trees Example
and 3 ways of adjusting it

Mini Binary Trees



```
class Tree {
  const left, right: Tree;
}
proc make(const depth: int): Tree {
  if depth <= 0 then
    return new Tree();
  else
    return new Tree(make(depth-1), make(depth-1));
}
proc free(const T: Tree) {
  if (T.left!=nil) {
    free(T.left); free(T.right);
  }
  delete T;
}
const T = make(5);
free(T);
```

What will I have to change?

```
class Tree {
  const left, right: Tree;
}
proc make(const depth: int): Tree {
  if depth <= 0 then
    return new Tree();
  else
    return new Tree(make(depth-1), make(depth-1));
}
proc free(const T: Tree) {
  if (T.left!=nil) {
    free(T.left); free(T.right);
  }
  delete T;
}
const T = make(5);
free(T);
```

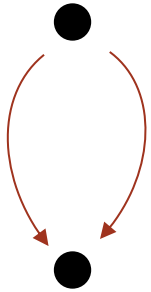
Using unmanaged for minimal changes

```
class Tree {
  const left, right: unmanaged Tree;
}
proc make(const depth: int): unmanaged Tree {
  if depth <= 0 then
    return new unmanaged Tree();
  else
    return new unmanaged Tree(make(depth-1), make(depth-1));
}
proc free(const T: unmanaged Tree) {
  if (T.left!=nil) {
    free(T.left); free(T.right);
  }
  delete T;
}
const T = make(5);
free(T);
```

Using owned to simplify this example

```
class Tree {
  const left, right: owned Tree;
}
proc make(const depth: int): owned Tree {
  if depth <= 0 then
    return new owned Tree();
  else
    return new owned Tree(make(depth-1), make(depth-1));
}
const T = make(5);
// now T will be destroyed when it goes out of scope
// and the free method is no longer necessary
```

Using shared in an acyclic graph



```
class DAG {
  const left, right: shared DAG;
}
proc make(const depth: int): shared DAG {
  // create a simple graph that refers to a node
  // multiple times
  var common = new shared DAG();
  return new shared DAG(common, common);
}
const T = make(5);
// now T will be destroyed when it goes out of scope
// and the free method is no longer necessary
```

Details

More About Borrowed

- **What is a borrow?**
  - a pointer to a class instance
  - that does not impact the lifetime of the instance
- **Class types default to 'borrowed'**
  - 'MyClass' is the same as 'borrowed MyClass'
  - Expect that borrowed is appropriate for most uses of classes
- **Several ways to borrow from a managed class value:**

```
class MyClass { ... }
var x = new owned MyClass();
// The following are equivalent ways of declaring a borrow from x:
var b = x.borrow();
var b: MyClass = x.borrow();
var b = x: MyClass; // Cast to borrow
var b: MyClass = x; // Coerce to borrow
```

Borrowed Arguments Don't Impact Lifetime

- An argument with borrowed type does not impact lifetime
- For example:

```
var global: borrowed MyClass;
proc saveit(arg: borrowed MyClass) {
  global = arg; // Error! trying to store borrow from local 'x' into 'global'
  delete arg;   // Error! trying to delete a borrow
}
proc test() {
  var x = new owned MyClass();
  saveit(x); // x coerced to borrow on call
  // instance destroyed here
}
test();
writeln(global);   // uh-oh! use-after free
```

Owned/Shared Arguments Impact Lifetime

- A default-intent 'owned' argument transfers ownership
- For example:

```
var global: owned MyClass;
proc saveit(arg: owned MyClass) {
  global = arg; // OK! Transfers ownership from 'arg' to 'global'
  // now instance will be deleted at end of program
}
proc test() {
  var x = new owned MyClass();
  saveit(x); // leaves x 'nil' - instance transferred to arg & then to global
  // instance not destroyed here since x is 'nil'
}
test();
writeln(global);   // OK
```

New Compile-time Checking

- **Lifetime checker is a new compiler component**
  - It checks that borrows do not outlive the relevant managed variable
- **For example, this will not compile:**

```
proc test() {
  var a: owned MyClass = new owned MyClass();
  // the instance referred to by a is deleted at end of scope
  var c: MyClass = a.borrow();
  // c "borrows" to the instance managed by a
  return c;  // lifetime checker error! returning borrow from local variable
  // a is deleted here
}

$ chpl ex.chpl --lifetime-checking
ex.chpl:1: In function 'test':
ex.chpl:6: error: Scoped variable c cannot be returned
ex.chpl:2: note: consider scope of a
```

Thanks!

```
class Tree {
  const left, right: owned Tree;
}
proc make(const depth: int): owned Tree {
  if depth <= 0 then
    return new owned Tree();
  else
    return new owned Tree(make(depth-1), make(depth-1));
}
const T = make(5);
// now T will be destroyed when it goes out of scope
// and the free method is no longer necessary
```

Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.

CRAY

THE SUPERCOMPUTER COMPANY