Language Improvements

Chapel Team, Cray Inc.

Chapel version 1.10

October 2nd, 2014

COMPUTE | STORE | ANALYZE

Safe Harbor Statement

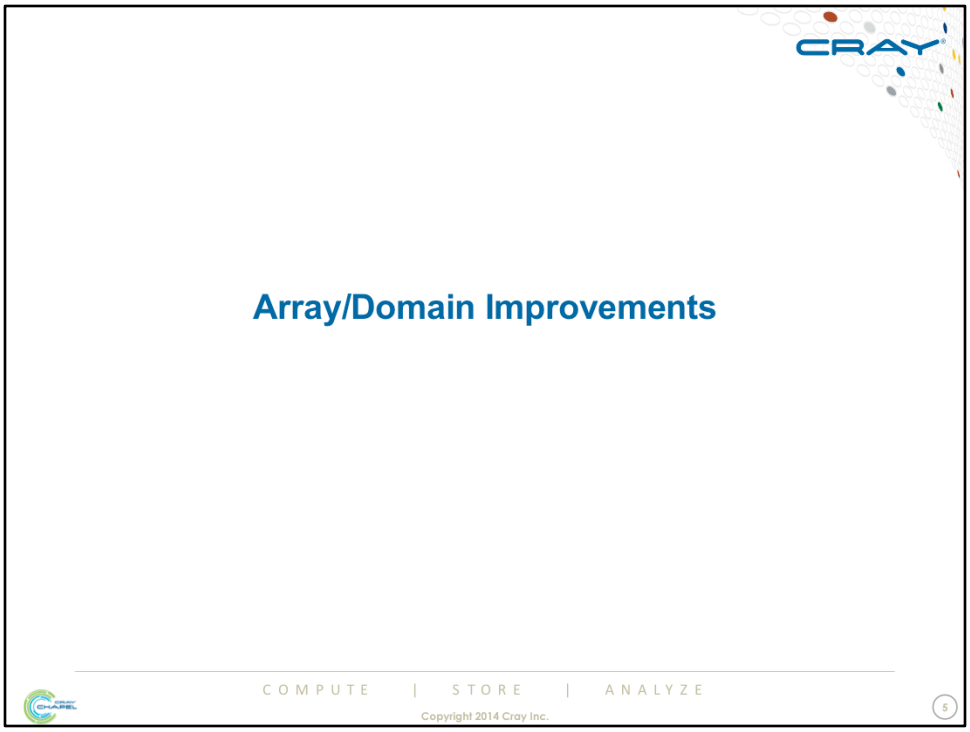
This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction 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.

Executive Summary

- **Language Improvements for this release include:**
  - improving common use cases for arrays/domains
    - based on FAQs and early user feedback
  - improving strings and constructors/initialization
    - long known to be weak points in Chapel for broad use
  - other changes geared at regularizing the language

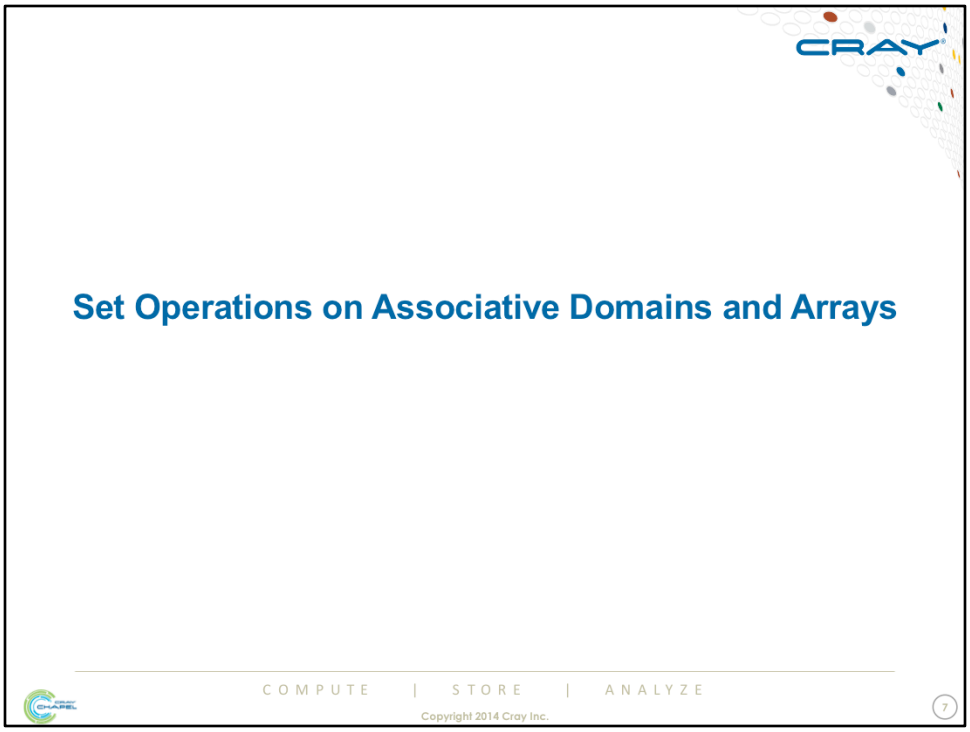
Outline

- Array/Domain Improvements
- String Improvements
- Semantic Improvements
- Syntactic Improvements
- Other Language Improvements


Array/Domain Improvements
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
5

Outline

- **Array/Domain Improvements**
  - Set Operations on Associative Domains/Arrays
  - Extend-on-write for Associative Arrays
  - Vector Operations on 1D Arrays
  - Querying a Locale's Subdomain
  - Pass Arrays to Extern Functions
- **String Improvements**
- **Semantic Improvements**
- **Syntactic Improvements**
- **Other Language Improvements**


Set Operations on Associative Domains and Arrays
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
7

Associative Set Operations: Background

- **Associative domains are sets at their core**
  - An associative domain is a collection of distinct indices
  - Simple addition and removal of indices has been supported

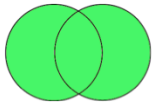
```
var a : domain(string);
a.add("Bob");
a.add("Alice");
```

- **Associative array operators did surprising things**
  - Associative array addition resulted in a rectangular array
  - And one with nondeterministic results at that

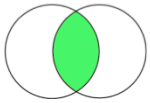
```
var a = ["a" => 1, "b" => 2];
var b = ["b" => 0, "c" => 3];
var c = a + b;
// c evaluates to [1, 5] or [2, 4], depending on how the unordered sets were zippered
```

Associative Set Operations: This Effort

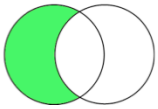
- **Support set operations on associative domains and arrays**
  - Union
    - + , +=, | , |=



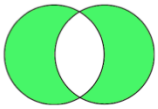
  - Intersection
    - & , &=



  - Difference
    - - , -=

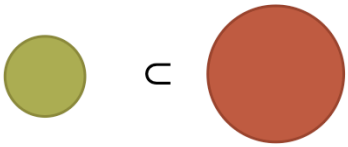


  - Symmetric Difference
    - ^ , ^=

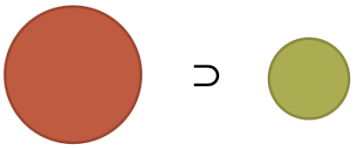


Associative Set Operations: This Effort

- IsSubset



- IsSuper



Associative Set Operations: Impact

- **Set operations are supported out of the box**
  - Domains

```
var a = { "a", "b", "c" };
var b =       { "b", "c", "d" };
var c = a & b;
// c evaluates to: { "b", "c" }
```

Associative Set Operations: Impact

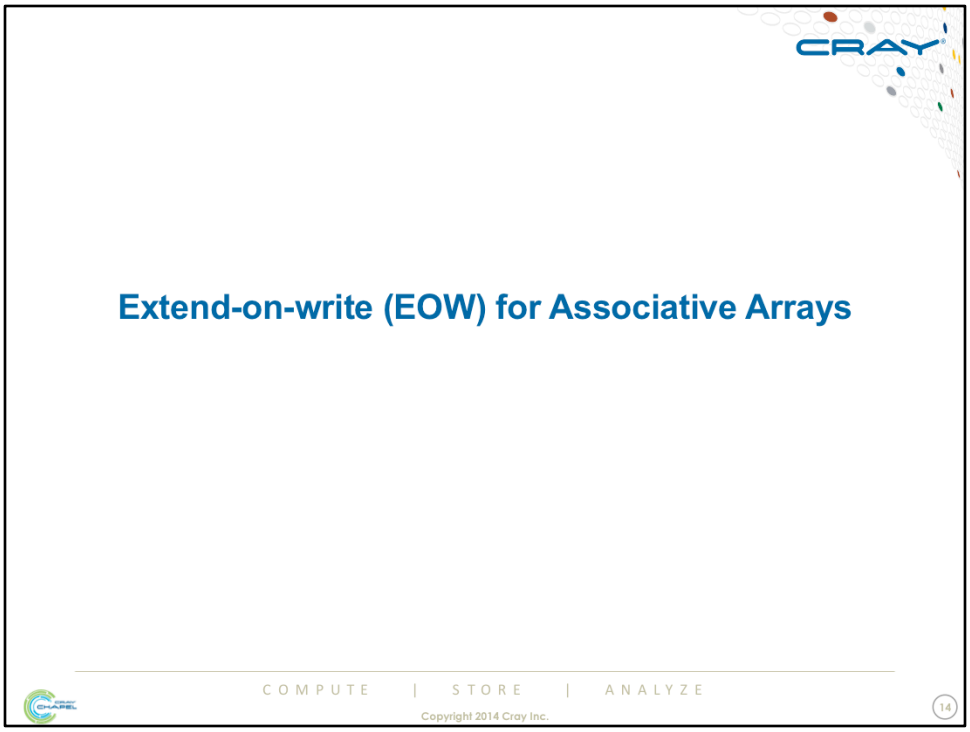
- **Associative array operations return associative arrays**
  - Set operations are restricted to arrays that don't share their domains

- **Unions**
  - When indices overlap, the second array's values take precedence

```
var a = ["a" => 1, "b" => 2];
var b = ["b" => 0, "c" => 3];
var c = a | b;
// c evaluates to: [ "a" => 1, "b" => 0, "c" => 3];
```

Associative Set Operations: Next Steps

- **Improve zippered iteration/promotion semantics for associative domains and arrays**
  - This effort has fixed some common/obvious/important cases
  - Yet, other functions should promote in a reasonable manner


Extend-on-write (EOW) for Associative Arrays
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
14

EOW for Associative Arrays: Background

- To extend an associative array, must modify its domain

```
var dom : domain(string);
var data : [dom] int;

dom.add("Bob");
data["Bob"] += 1;
```

- Accessing an associative array at other indices is an error

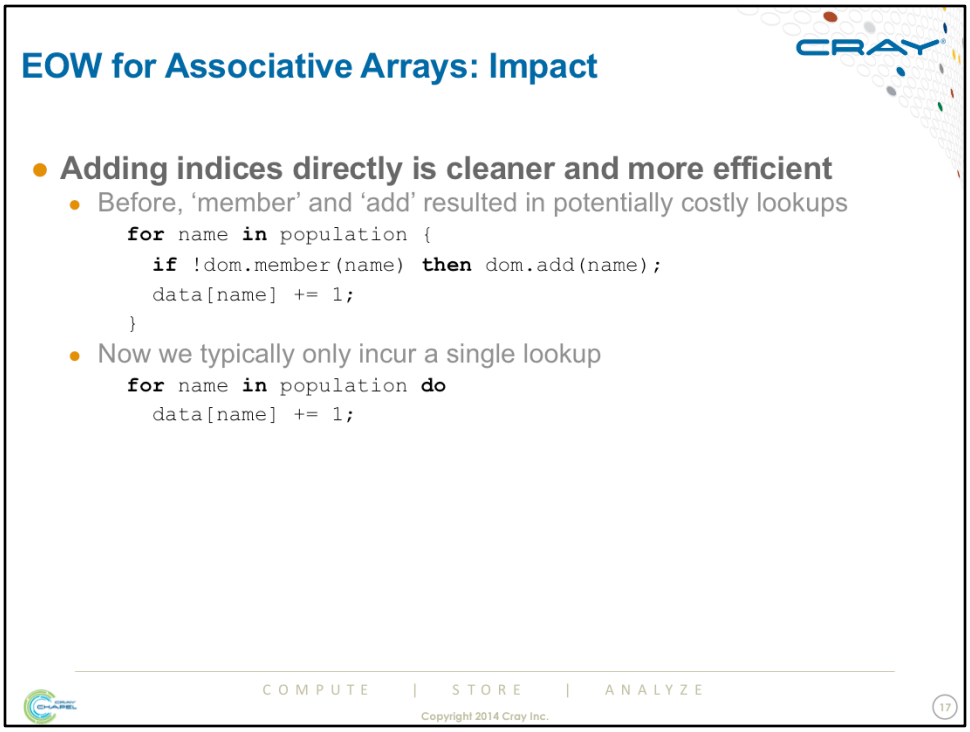
```
data["Doug"] += 1;   // results in an out-of-bounds error
```

EOW for Associative Arrays: This Effort

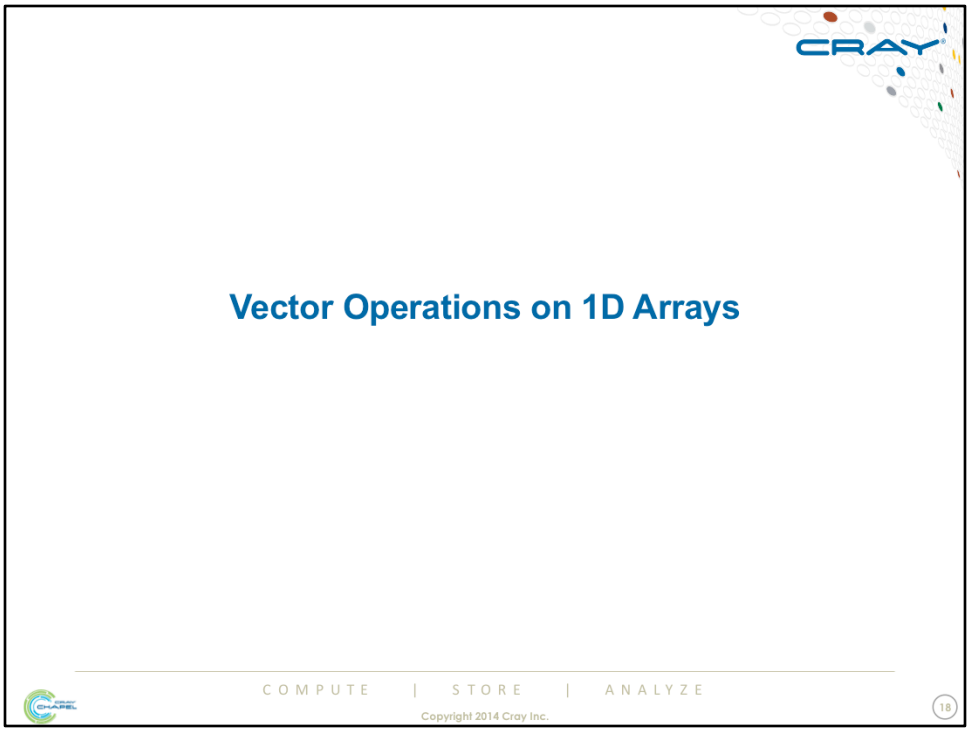
- **We now support extension of associative arrays via writes**
  - Restriction: array must not share its domain

```
var dom : domain(string);
var data : [dom] int;
data["Bob"] += 1;
…dom.member("Bob")…   // evalutes to "true"

data["Doug"] += 1;
…dom.member("Doug")… // evaluates to "true"
```



The "typically" qualification in the final bullet refers to the possibility of requiring multiple lookups if other tasks are simultaneously modifying the associative domain/ array


Vector Operations on 1D Arrays
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
18

Vector ops on 1D arrays: Background

- **Want vector type that supports adding/removing elements**
  - Support operations similar to those available in
    - Python's 'array' type
    - C++'s std::vector
  - Rather than defining a Vector class, add the operations to 1D arrays

- **Before, changing an array's size required explicit domains**

```
var D = {1..5};
var A: [D] int = [i in D] i;
D = {1..6}; // extend D in order to add an element to A
A[6] = 6;
```

Vector ops on 1D arrays: This Effort

- **Operations that modify the array's domain**

  push_back(val), pop_back()
  - Add or remove an element at the back of the array
  - Extends or reduces the domain by one index at the back

  push_front(val), pop_front()
  - Add or remove a value at the front of the array
  - Extends or reduces the domain by one index at the front

  insert(pos,val), remove(pos), remove(rng),remove(pos,cnt)
  - Insert or remove values from arbitrary positions
  - Extends/reduces domain by the number of values inserted/ removed
  - Modifies the domain from the high end and shifts array elements

  clear()
  - Remove all values from the array and leave the domain empty

Vector ops on 1D arrays: This Effort

- **Operations that do not modify the array's domain**

  head(), tail()
  - Return the first or last element in the array

  isEmpty()
  - Return 'true' if the array has no elements, 'false' otherwise

  find(val)
  - Return a tuple containing a boolean and an index
  - If the boolean is true, the value was found at the index

  count(val)
  - Return the number of times 'val' occurs in the array

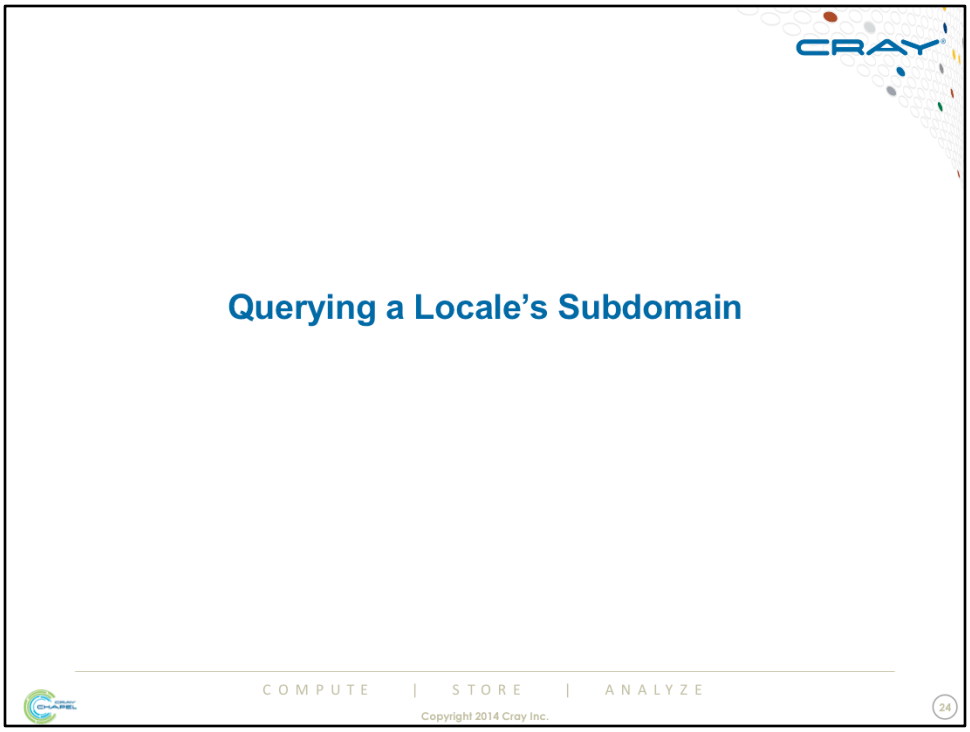
  reverse()
  - Reverse the order of the elements in the array

Vector ops on 1D arrays: Caveats

- **Supported only for arrays that do not share their domain**
  - Otherwise modifying the domain could cause multiple arrays to update
    - (considered too confusing to support by default)
  - Checked at runtime
- **Methods that modify domains always cause reallocations**
  - Not extremely costly for small arrays
  - Use sparingly on larger arrays
- **For detailed examples, see the primer at:**
  - $CHPL_HOME/examples/primers/arrayVectorOps.chpl

Vector ops on 1D arrays: Next steps

- **Optimize to amortize reallocation costs**
  - A recursive doubling/halving scheme would improve push/pop ops
    - But requires maintaining runtime state that typical 1D arrays don't need
- **Refine interface**
  - Add additional methods if desired
  - Consider changes to method names based on user feedback


Querying a Locale's Subdomain
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
24

Querying a Locale's Subdomain: Background

- Domain maps distribute index sets to locales
- Determining a locale's local index set was challenging
  - Required using non-public/developer interfaces

```
var DM = Dom dmapped Block(Dom);
var Data : [DM] int;

on Locales[1] {
  var myIndices = Data._value…
}
```

Querying a Locale's Subdomain: This Effort

- **Support for querying these sets through arrays**

hasSingleLocalSubdomain()
- Can the local index set be represented by a single domain?

localSubdomain()
- If possible, will return a single subdomain representing the set

localSubdomains()
- An iterator; yields as many domains as necessary to represent the set

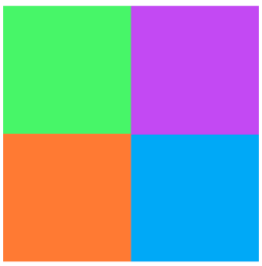
targetLocales()
- The array of locales targeted by a domain map

Querying a Locale's Subdomain: Impact

- Now possible to use the index set in computations

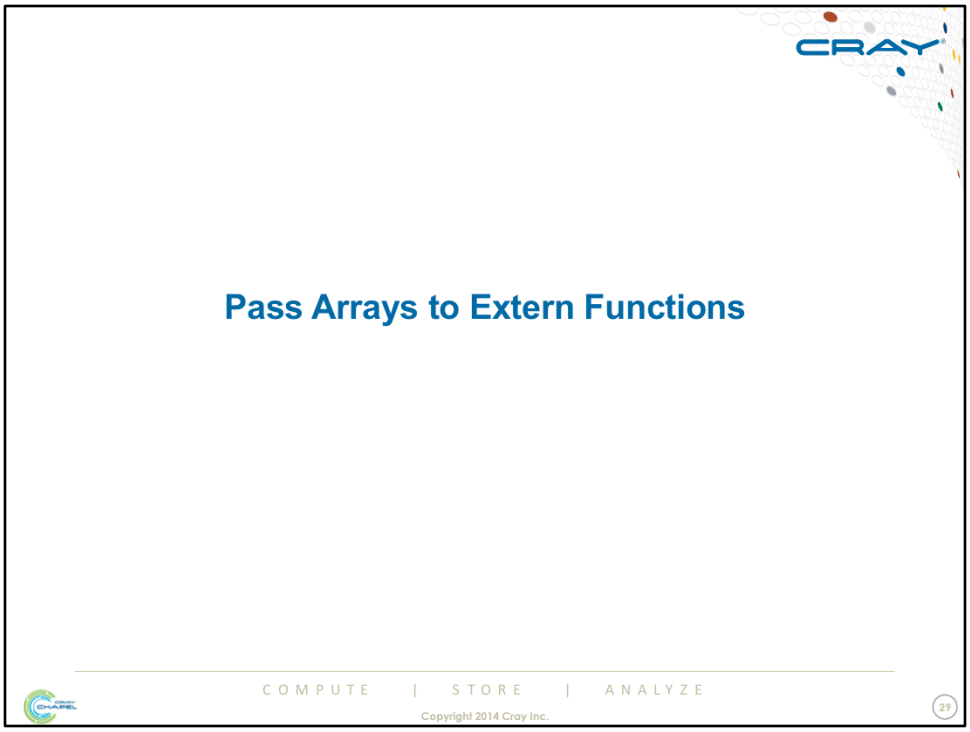
```
var Space = {1..n};
var DS = Space dmapped Block(Space);
var Colors : [DS] Color;

for L in Colors.targetLocales() do
  on L do
    Colors[Colors.localSubdomain()] = nextColor();
```



Querying a Locale's Subdomain: Next Steps

- Support for similar queries on domains and domain maps
- Parallel 'localSubdomains()' iterator (?)
- Refine naming based on feedback (?)


Pass Arrays to Extern Functions
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
29

Pass Arrays to Extern Functions

Background: Passing arrays to extern functions was ugly

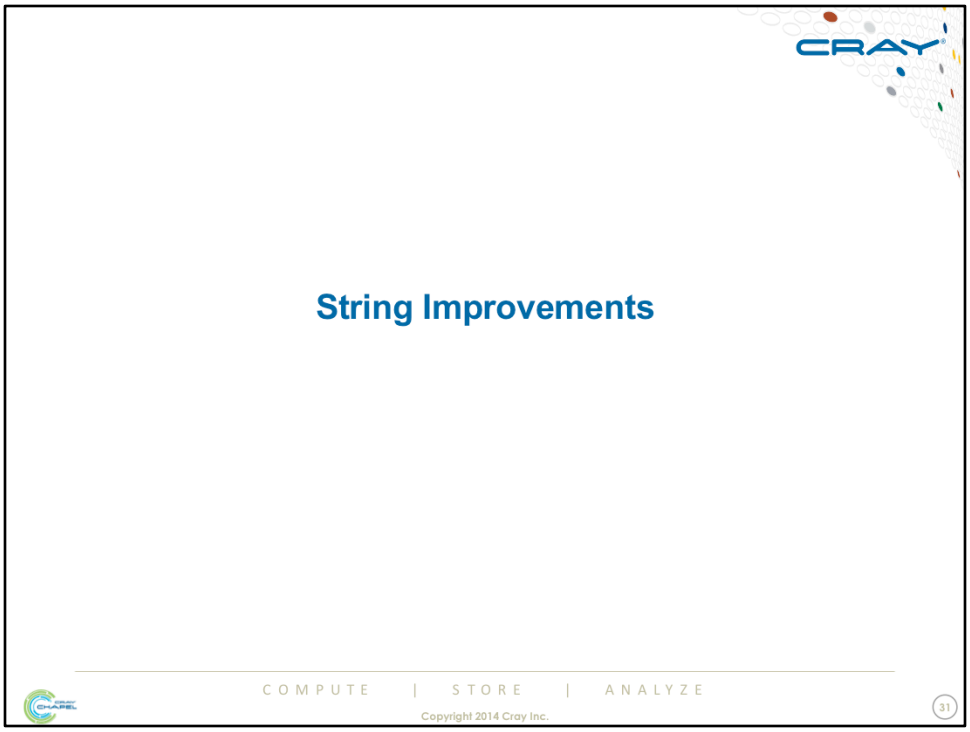
```
extern proc print_array(ref x: int, n: int);
var A = [1, 2, 3, 4];
print_array(A[1], A.size);
```

This Effort: Support extern procedures that accept arrays

```
extern proc print_array(x: [] int, n: int);
var A = [1, 2, 3, 4];
print_array(A, A.size);
```

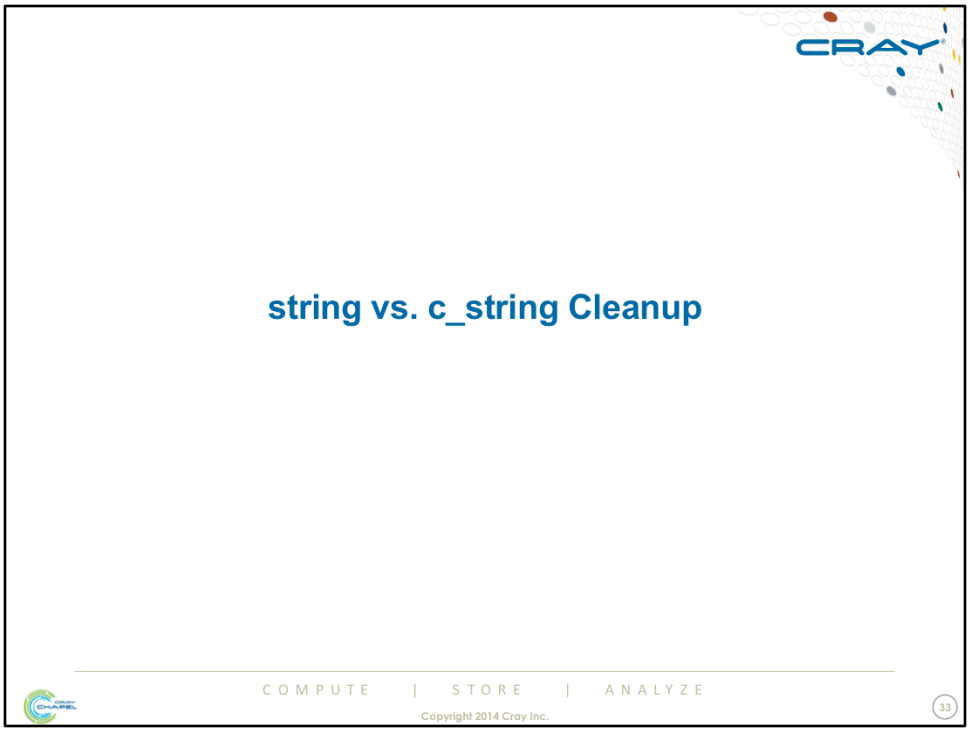
Next Steps: Support additional array types

- Currently only works with 1D arrays
- We should be able to support passing any contiguous array to C
  - e.g., non-distributed *n*-dimensional arrays


String Improvements
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
31

Outline

- Array/Domain Improvements
- String Improvements
  - string vs. c_string Cleanup
  - Leak-free String Implementation
- Semantic Improvements
- Syntactic Improvements
- Other Language Improvements


string vs. c_string Cleanup
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
33

string vs. c_string: Background

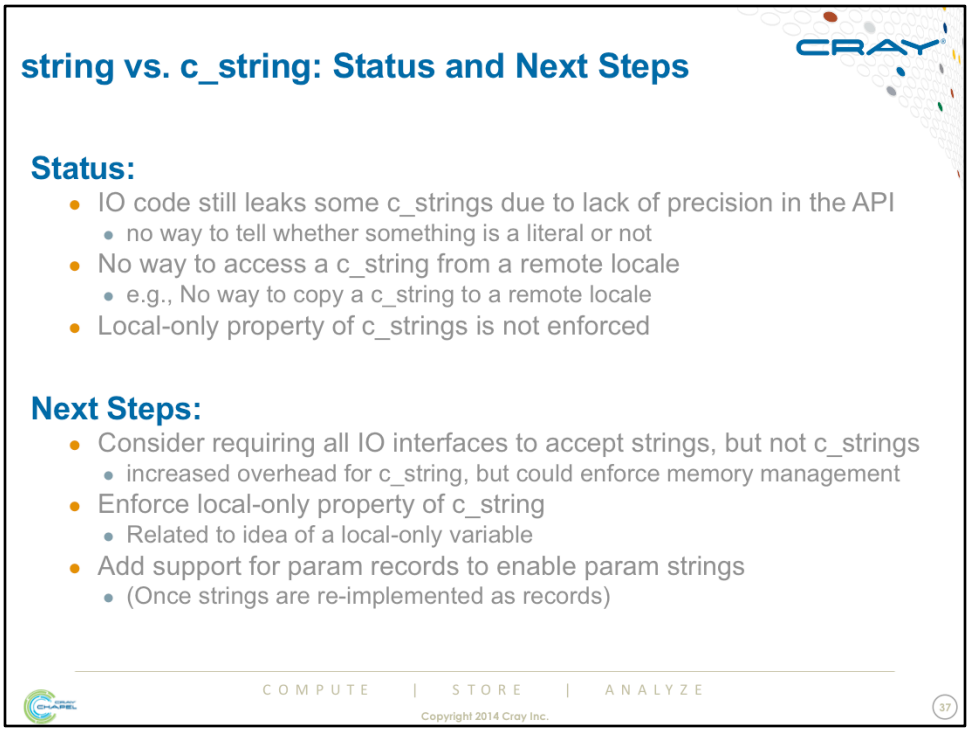
- **Added c_string type for use with external C functions**
  - c_string is considered a local type
    - i.e., should never be remote, though the compiler does not enforce this
  - Chapel strings can be passed to external C functions using .c_str()
  - c_strings can be turned into Chapel strings using toString()
    - or by simply using the cast operator
- **For convenience, both types were used interchangeably**
  - Led to increased memory leaks and other undesirable behavior
    - particularly when used in a multilocale setting
  - Made it difficult to trivially change the Chapel string implementation
- **Have been working on making Chapel strings less special**
  - Today, it's a special type in the compiler and runtime
  - Working toward implementing strings as a Chapel record
    - to remove special cases for them in the code base
    - to close string-related memory leaks

string vs. c_string: This Effort

- **Distinguish between the Chapel strings and c_strings**
  - Made all extern interfaces use c_string
  - Removed (almost) all knowledge of Chapel strings in the runtime
  - Made all string literals in the language be of type c_string
    - added automatic coercion to Chapel string from c_string literal
    - removed param Chapel strings in favor of param c_string literals
- **Provide a Chapel function to free c_strings**
  - Libraries using c_string types, namely I/O, can clean up c_strings
  - Most users should just use Chapel strings and will not need this

string vs. c_string: Impact

- **Almost all runtime functions now operate on c_string**
  - Facilitates drop-in replacements for the Chapel string implementation
  - Few remaining cases support remote strings
    - these will go away once strings are records
- **Assorted clean-up**
  - Moved Chapel string and c_string types into separate modules
    - Many Chapel string functions are implemented in terms of c_string functions, some of which call extern C functions
  - Removed “string_normalize” primitive
    - (used to recalculate length of string returned from extern C function)
  - Plugged c_string leaks in IO code when possible



With respect to "No way to copy a c_string to a remote locale": It can be done by abusing an incorrect behavior in the compiler (see MemTracking.chpl) or when passing strings by reference. Both will go away with new string implementation.


Leak-free String Implementation
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
38

Leak-free strings: Background

- **Chapel strings are implemented as C character arrays**
  - Heap allocated
  - Mostly never copied to other locales
- **Chapel strings are leaked**
  - Assignment is by reference and strings aren't reference counted
  - This has been a longstanding wart in the Chapel implementation

Leak-free strings: This Effort

- **Implement a leak-free record-based string type in Chapel**
  - Mutable, copied by value
  - Provide all the current string operations
    - minus a couple that we decided to abandon

```
record string_rec {
  var base: baseType;          // c_string
  var len: int;
  var home: locale;            // where string is physically allocated
  var refCnt: string_refcnt;   // wrapped ref count class
  var aliasRefCnt;             // ref count if aliased (via autocopy)
}
```

  - All remote copying of the base string is handled by the module
  - Available in temporary standard module NewString
    - Next release: replace 'string' with a variation on this implementation

Leak-free strings: This Effort

- **New test suite to demonstrate leak-free operation**
  - Generalized test suite to support drop-in string implementations
    - works with current Chapel strings and string_rec, should work for others
  - Utility functions for tracking memory leaks
    - Single- and multi-locale support
    - possibly pull out for common use
  - Current suite covers:
    - initcopy (copy constructor)
    - assignment
    - cast
    - concatenation
    - ascii
    - find (was indexOf)
    - substring
    - relational operators
    - Other Chapel concepts: begin, coforall, promotion
  - Available in test/types/strings/StringImpl

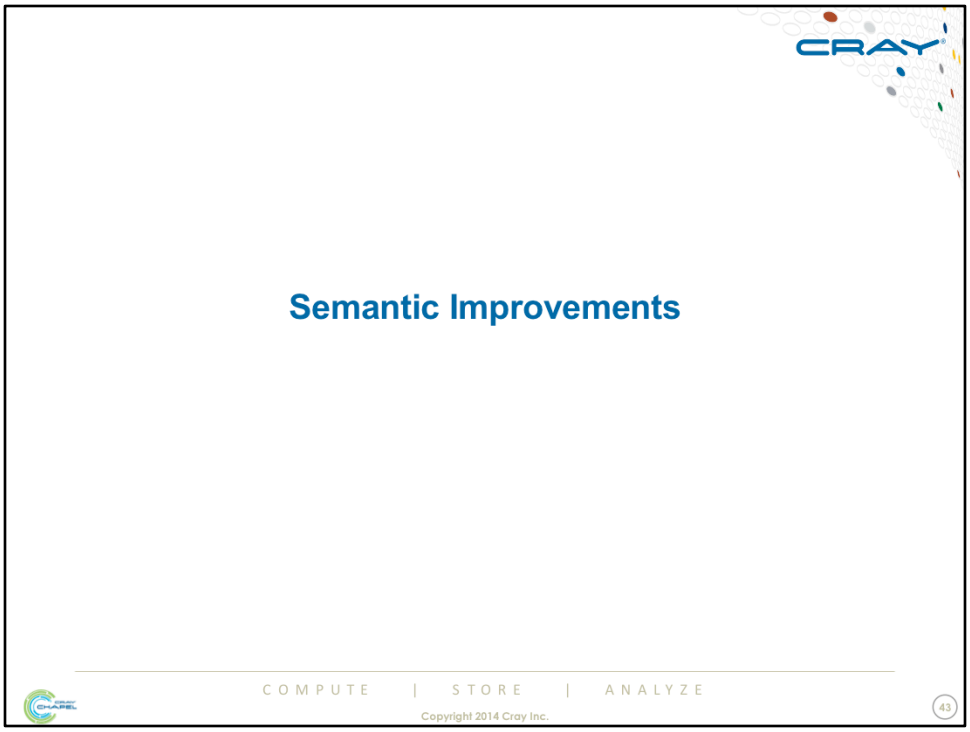
Leak-free strings: Status and Next Steps

Status:

- Leak-free string implementation using a record with reference counting
  - passes all single- and multi-locale StringImpl tests

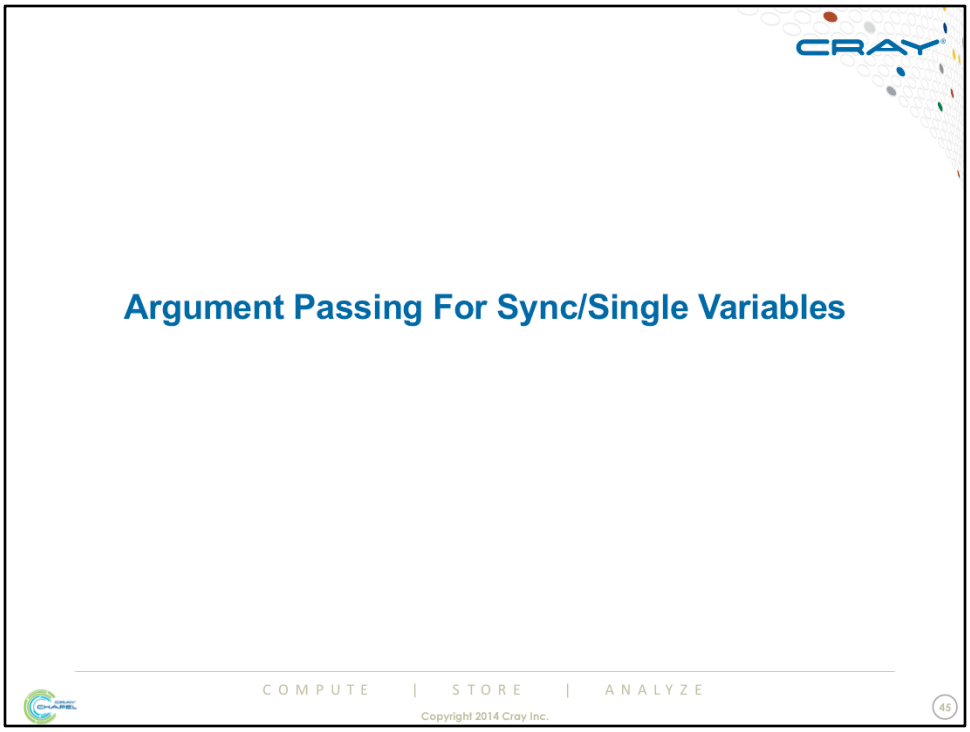
Next Steps:

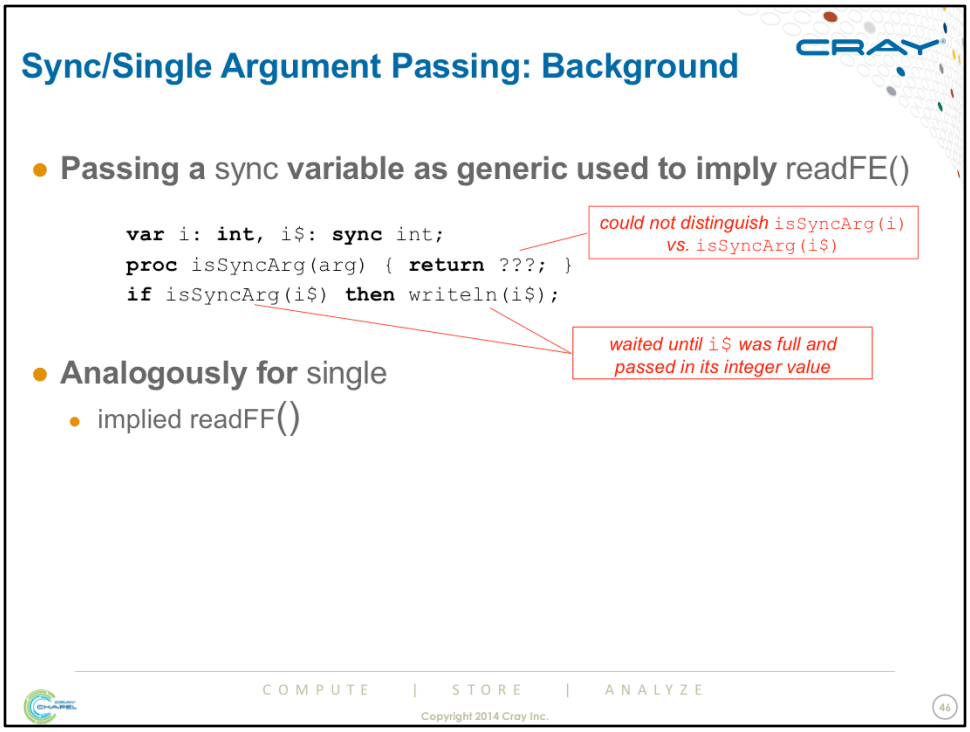
- Replace existing string implementation with string_rec
- Clean up vestiges of current string implementation
  - Remove special case code for wide strings in the compiler
  - Remove runtime support for wide strings
  - Clean up other string-specific code, e.g., primitives
- Extend StringImpl test suite to include performance tests
- Consider refining string_rec:
  - Should we remove the mutability quality?
  - Must string_recs be reference counted?
- Make strings UTF-8
- Implement a more complete string library, similar to Python's/C++'s


Semantic Improvements
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
43

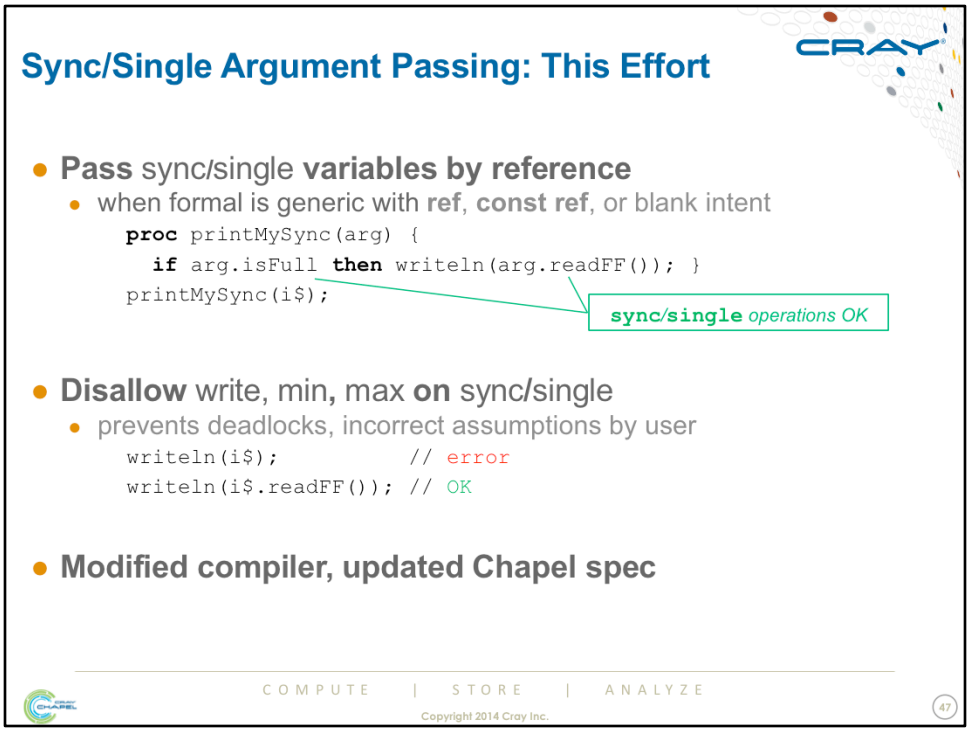
Outline

- Array/Domain Improvements
- String Improvements
- Semantic Improvements
  - Argument Passing for Sync/Single Variables
  - Reference Variables
  - Default Initialization
  - Noinit
  - Call User-Defined Default Constructors
- Syntactic Improvements
- Other Language Improvements


Argument Passing For Sync/Single Variables
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
45



The implicit addition of readFE() for sync vars or readFF() for single vars was inconsistent with how other types are passed to formals of generic type.

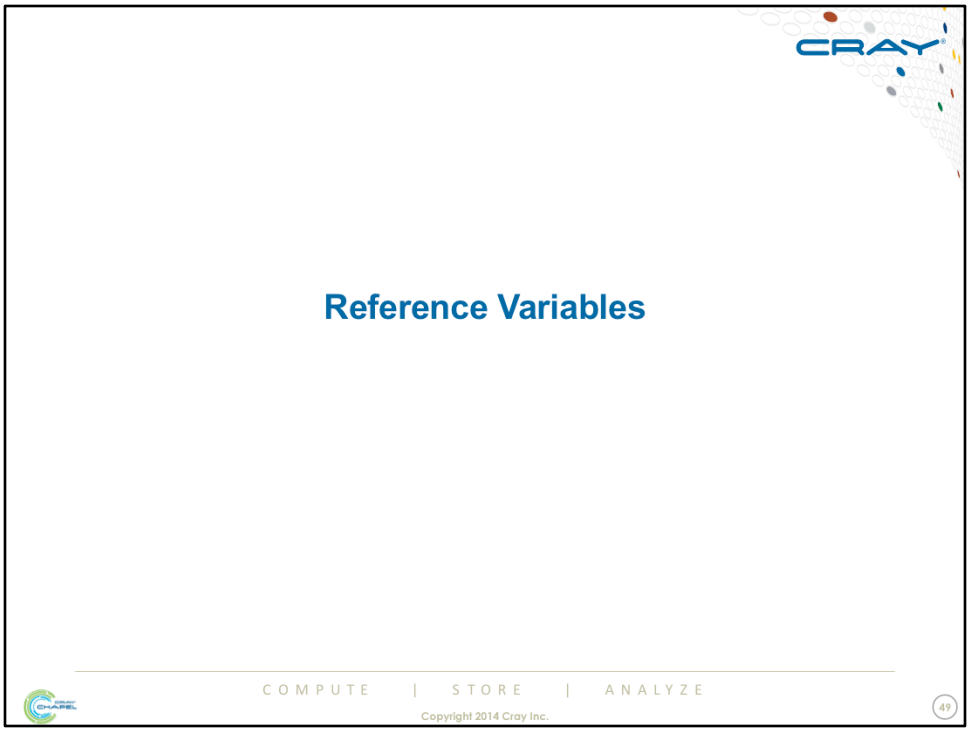


readFE()/readFF() continue to be inserted implicitly when the actual is sync/single and the corresponding formal is of a primitive type.

When the formal argument with ref, const ref, or blank intent has an explicit sync/ single type, the actual argument is also passed by reference – this has been the case before and is not affected by this change.

Sync/Single Argument Passing: Next Steps

- **Extend argument passing rule to other intents**
  - e.g., in, out
- **Seeking community input on** read(), write()
  - what should happen for **write(i$)**?
    - perform **i$.readFE()** (former behavior)
    - disallow; user must extract value as desired (current behavior)
    - output whether **i$** is full or empty and its value (if full)
    - other?
  - what should happen for **write(r)** when r is a record or class *with a* **sync**/**single** *field*?
    - do **write()** on the field, as above
      currently forces user to implement **writeThis()**
    - treat the field specially, e.g. **write(field.readXX())**
  - should **read()** be symmetric to **write()** ?


Reference Variables
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
49

Reference Variables: Background

- **Currently a mismatch in support for references**
  - Possible to write a function that captures a reference

```
proc update(ref x) {
    ...
    x = foo(x);
}
update(my_big_data_structure.access(i));
```

  - Yet, not possible to create a variable that captures a reference
    - Thus, to avoid helper functions, must write something like:

```
my_big_data_structure.access(i) =
  foo(my_big_data_structure.access(i));
```

    - This is both syntactically and computationally redundant

Reference Variables: This Effort and Impact

This Effort: Add initial support for reference variables

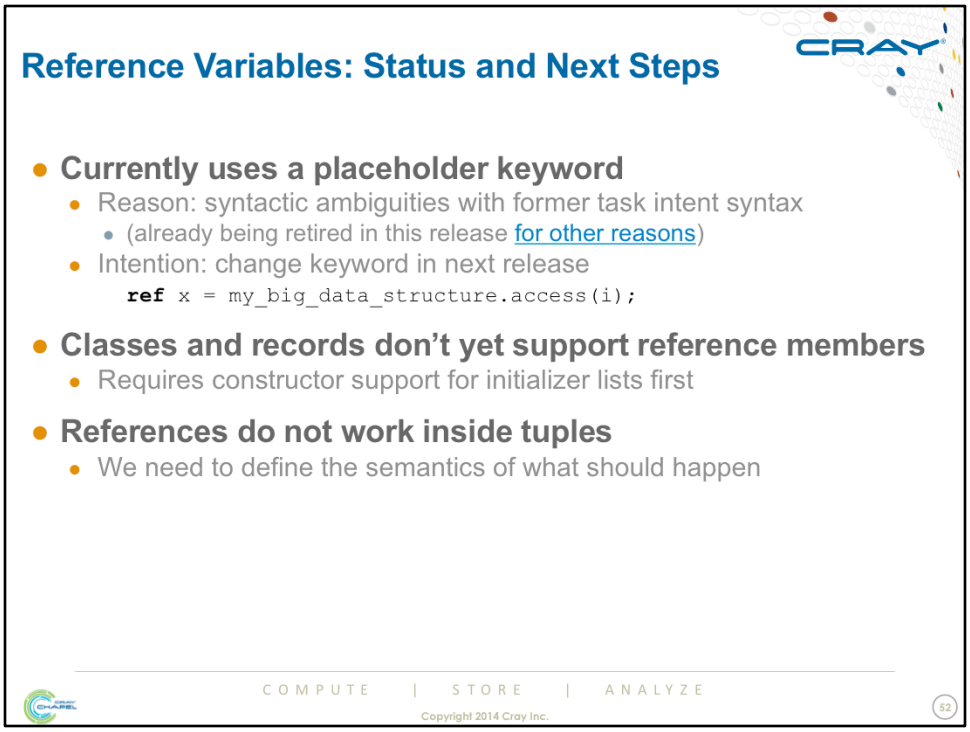
```
refvar x = my_big_data_structure.access(i);
...
x = foo(x);
```

- References must be initialized when they are defined
- All subsequent reads and writes are to the aliased value

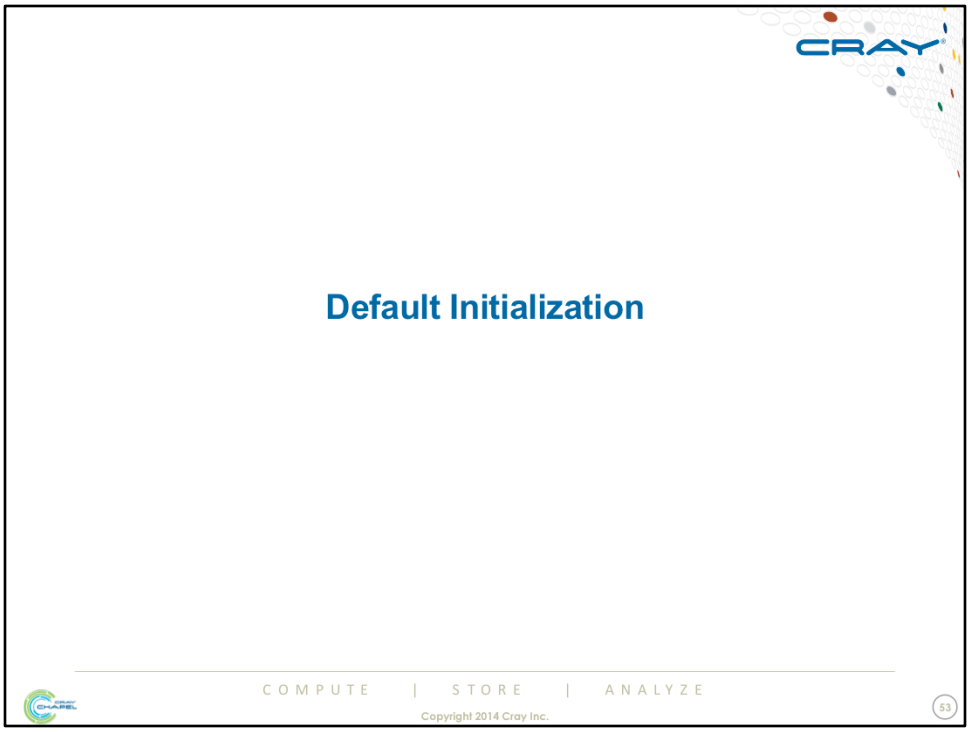
```
var a = 10;
refvar b = a;
b += 1;
writeln(a); // -> 11
a += 1;
writeln(b); // -> 12
```

Impact: Resolved mismatch in support for references

- No need to use a function to capture a reference



References cannot be members at the moment since they are required to be initialized at the definition point. Once Chapel has something analogous to initializer lists we can add support for them as members.


Default Initialization
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
53

Default Init

Background:

- Default values for types have been hard-coded into the compiler
  - Implemented via PRIM_INIT, involving lots of special-case code
  - Adding a new type required compiler modification for unusual defaults
    - Generally a complicated effort
    - This area of our compiler is not particularly friendly

This Effort:

- Default values can now be defined at module level
  - Mechanism: Overload a specifically-named function:

```
proc _defaultOf(type t) where (t == <type>) {…}
```

  - For example:

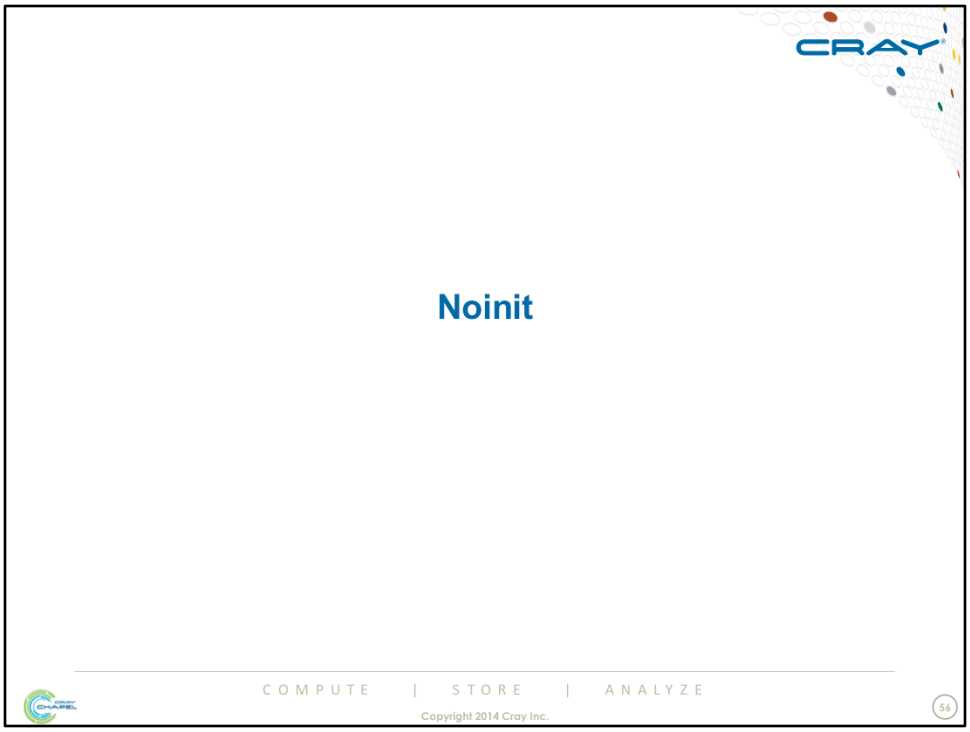
```
proc _defaultOf(type t) where (t == int) { return 0; }
```

Impact:

- Less special-case code in the compiler
- Type author can define defaults in Chapel code near type definition

Default Init: Next Steps

- **Implement _defaultOf() for remaining types**
  - distributions are the primary outlier
  - parallel default array initialization is another case that needs attention
- **Promote _defaultOf() to a supported user-level concept**
  - integrating it with the default constructor story
- **Replace remaining compiler instances of PRIM_INIT**
  - Some code simplification already done
  - Requires better separation of initialization from type determination
- **Remove Type::defaultValue from compiler**
  - This field is no longer necessary

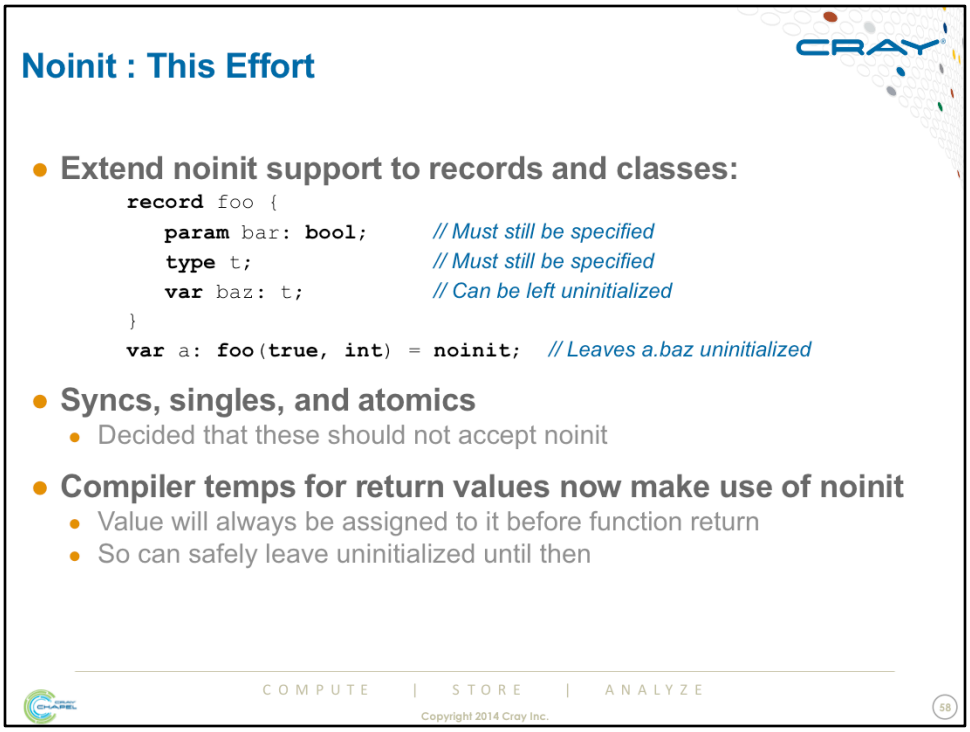

Noinit
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
56

Noinit and Default Initialization: Background

- **Version 1.9 introduced the keyword 'noinit'**

```
var foo: <type> = noinit;
```

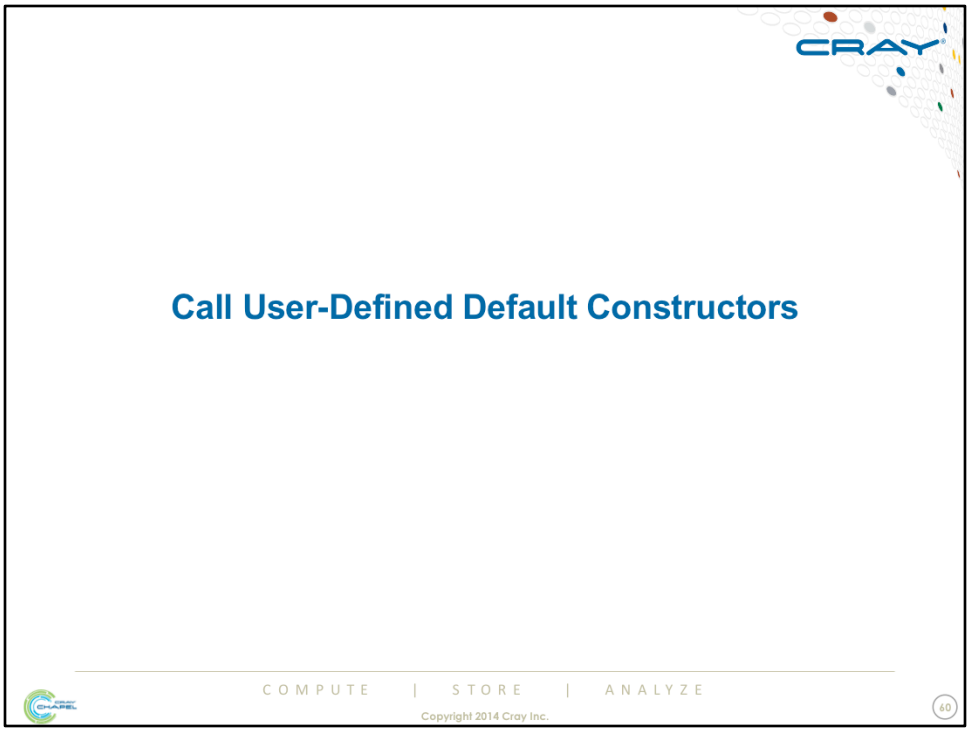
  - Permits a user to squash default initialization
    - Designed to support performance benefits, especially for arrays
  - Only implemented for primitive types as of version 1.9
    - ints, strings, bools, enums, ranges, etc.
    - Certain basic classes would work, but distinguishing which was costly

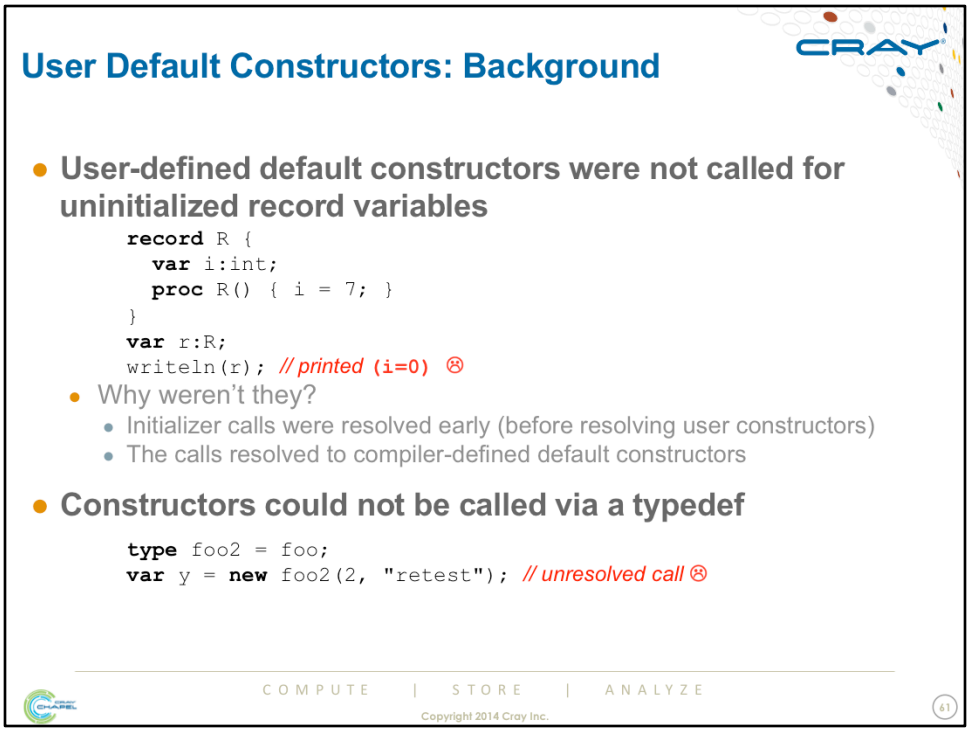


It was decided that an uninitialized state for syncs, singles, and atomics did not make sense.

Noinit: Next Steps

- **Extend noinit to remaining types**
  - domains and arrays are the major outstanding cases
- **Provide richer story for user-defined noinit on types**
  - Integrate with _defaultOf and constructor features
- **Identify cases in modules where noinit can be used**
  - Anticipated performance benefits
- **Use noinit with other compiler generated temp variables**

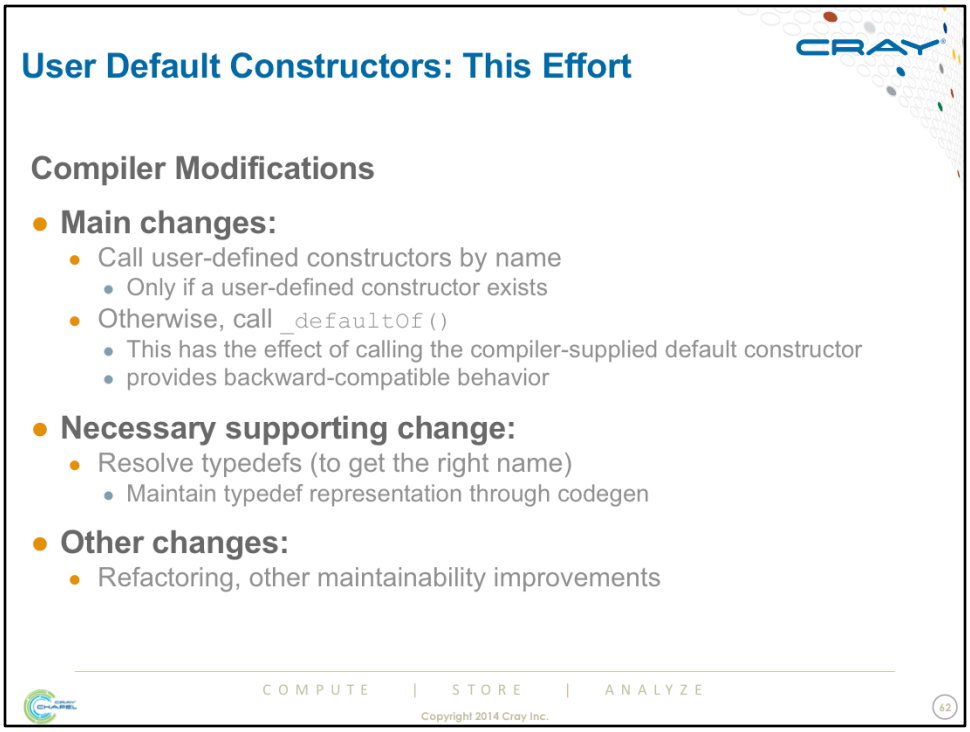

Call User-Defined Default Constructors
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
60



The problem with not calling a user-defined constructor is that the class designer cannot establish invariants in a default-initialized object.

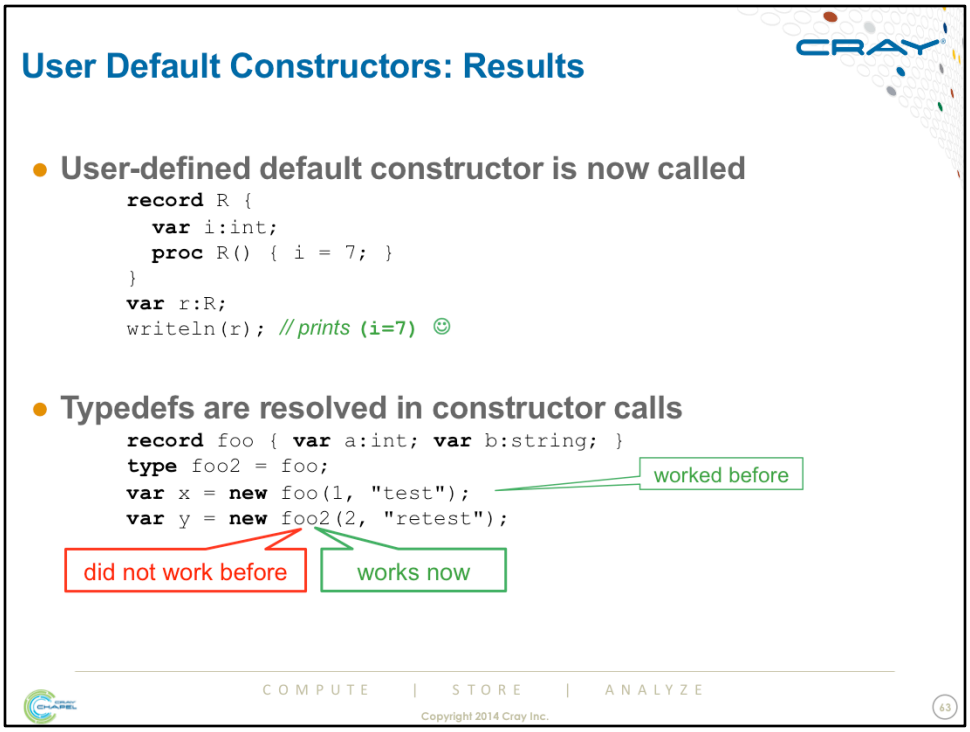
The compiler can't possibly know what those invariants are, so the compiler-supplied default initializer will not necessarily do the right thing.

For example in a reference-counted type, the invariant would be to set the initial reference count to 1. Default initialization would set it to zero.



Implementing calls to user-defined default constructors and the encapsulation of default-initialization behavior in the _defaultOf() function were separate efforts.

It was easy to integrate these, because we want to call the user-defined default constructor if it exists and otherwise use the compiler-supplied default constructor.



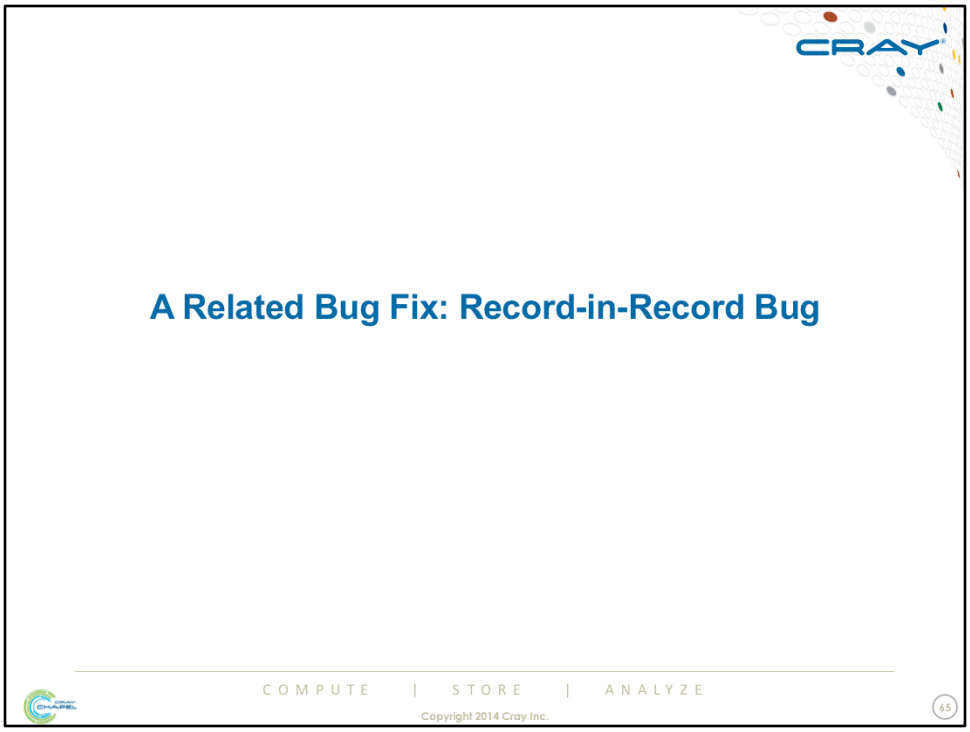
Before, an attempt to call a constructor using a typedef to name the type to be constructed resulted in an unresolved call error.

It should also be possible to use a type function in a constructor call (e.g. "new fnReturnsType()()"), but this does not currently work.

The concept is pretty powerful, because it directly supports the factory pattern.

User Default Constructors: Next Steps

- **Unify constructor features with** `_defaultOf()/noinit`
  - and promote to a concept for end-users
- **Extend construct-through-typedef to other cases**
  - e.g., functions returning types
- **Replace** `chpl__initCopy` **with constructor calls**
- **Convert constructors to methods**
- **Separate allocation from construction**
  - Enable reusing an already-allocated object
  - Support custom allocators
- **Add initializer lists to constructors**
  - Separates field initialization from assignment
  - Supports `ref` fields in class and record types


A Related Bug Fix: Record-in-Record Bug
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
65

Fix Record-in-Record Bug: Background

Background:

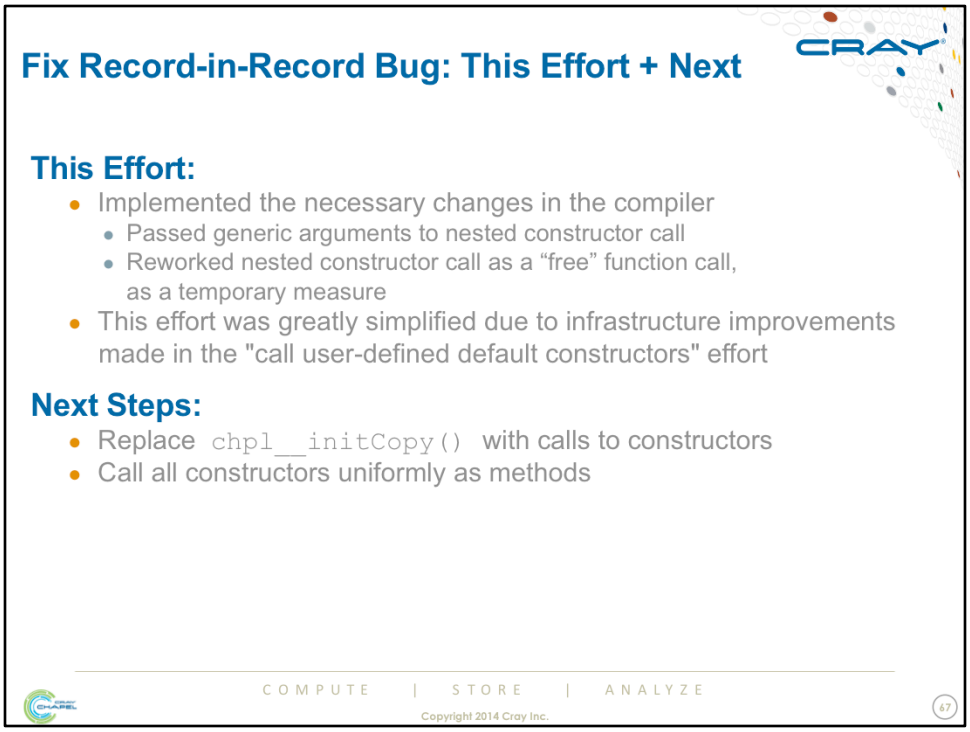
- Attempting to construct an instance of a type containing a nested record failed with an unresolved symbol error

```
record R { type t; var x:t;
  record R2 { var y:t; }
  proc test() { var r2:R2;
                writeln(r2.y); }
}
```

Failed here, attempting to create an R2

The Problem:

- Generic arguments were not being passed to nested constructor calls
- Nested constructors were being called as methods

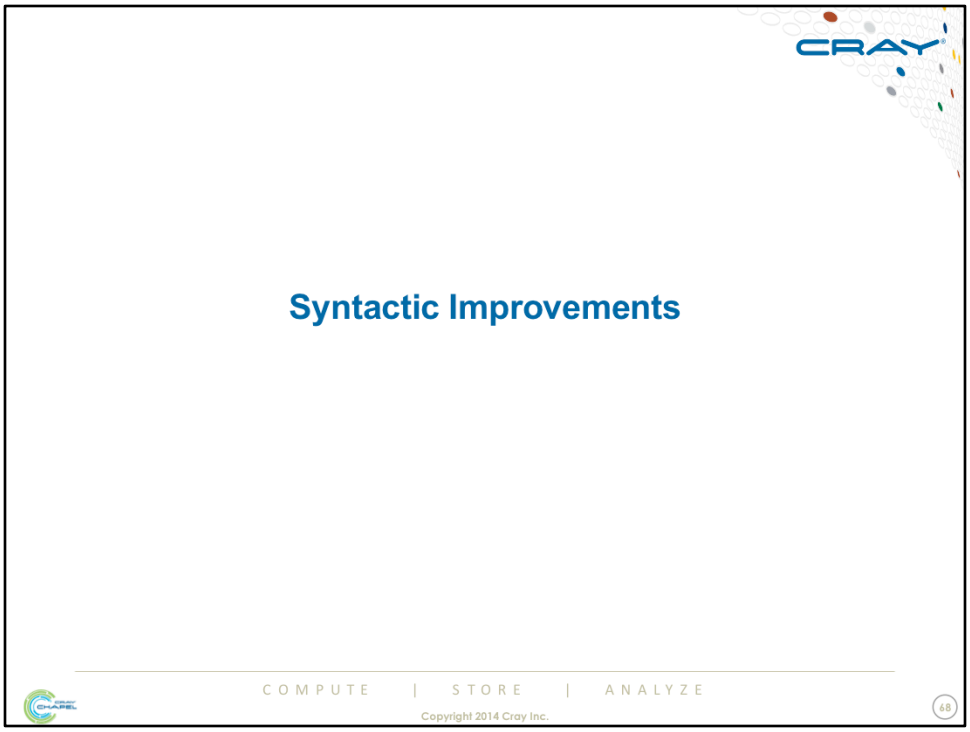


Calls being generated to invoke nested constructors were changed from method calls to "free" function calls. This is for uniformity, because constructors are currently implemented as "free" functions.

Turning all constructors into methods is planned as future work.

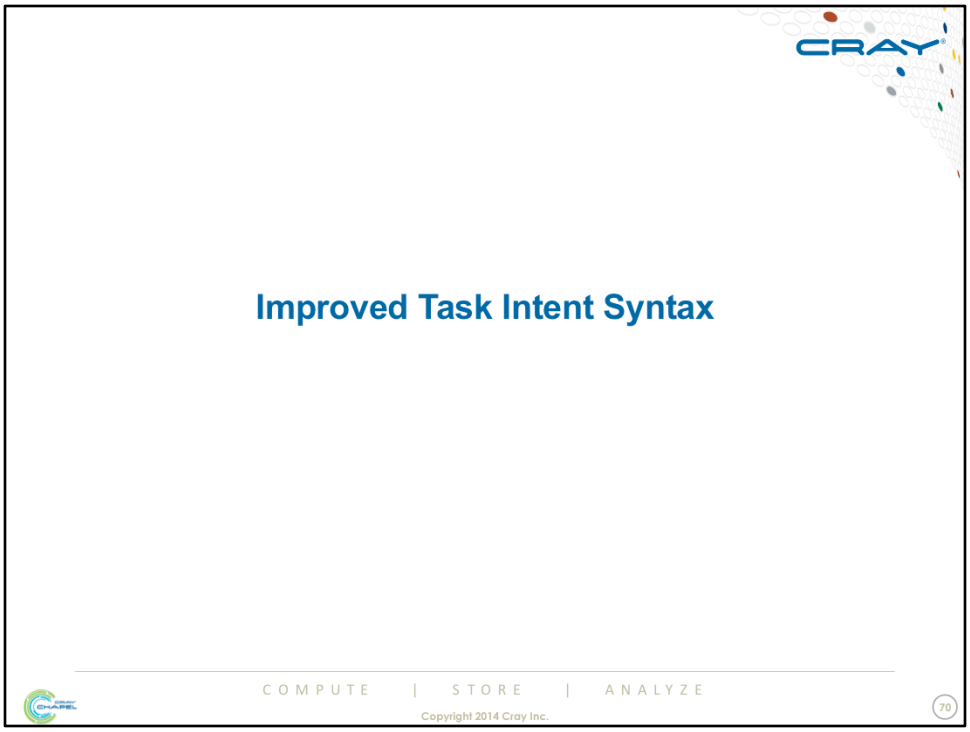
The modification to pass generic arguments to the nested constructor call is necessary regardless whether the constructor is called as a method or a free function, and will be retained when constructors are converted to methods.

Both pieces took only a few days to implement and commit. This is a noteworthy reward from the effort spent this release cycle to improve the maintainability of the compiler.


Syntactic Improvements
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
68

Outline

- Array/Domain Improvements
- String Improvements
- Semantic Improvements
- Syntactic Improvements
  - Improved Task Intent Syntax
  - Var Function Deprecation
- Other Language Improvements


Improved Task Intent Syntax
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
70

Task Intent Syntax: Background

- **default task intents prevent certain common data races**
  - e.g. shapshot *i* when **begin** task is created, rather than reading later

```
var i = 0;
while i < 10 {
  begin f(i);   // now guaranteed to see each of f(0), f(1), …, f(9)
  i += 1;
}
```

- **`ref` intent allows sharing variables between tasks**
  - e.g. producer-consumer

```
var lock$: sync int;
var data: Data;
cobegin ref(data) {
  while … { lock$=1; produce(data); lock$; }
  while … { lock$=1; consume(data); lock$; }
}
```

pass status *by* **ref** *intent* (old syntax)

Task Intent Syntax: Background

- **Want support for other argument intents as well**
  - e.g. 'in' intent to support task-private copy of variables

```
var i = …;
cobegin in(i); {
  for … { i += 1; …A[i]… }
  for … { i += 1; …B[i]… }
}
```

each task has its own private copy of **i**

- **But current syntax breaks down quickly**
  - (for humans, if not for parsers)
  - particularly with loop-based constructs

```
var i = …;
coforall j in D in(i) { … }
coforall D in(i) { … }
```

in is too overloaded in this context…

Task Intent Syntax: This Effort

- **Improved task intent syntax**
  - syntactically similar to a list of function formals
  - added a new keyword: `with`

```
var lock$: sync int;
var data: Data;
cobegin ref(data, something_else) {
cobegin with (ref data, ref something_else) {
  while … { lock$=1; produce(data); lock$; }
  while … { lock$=1; consume(data); lock$; }
}
```

old syntax

new syntax

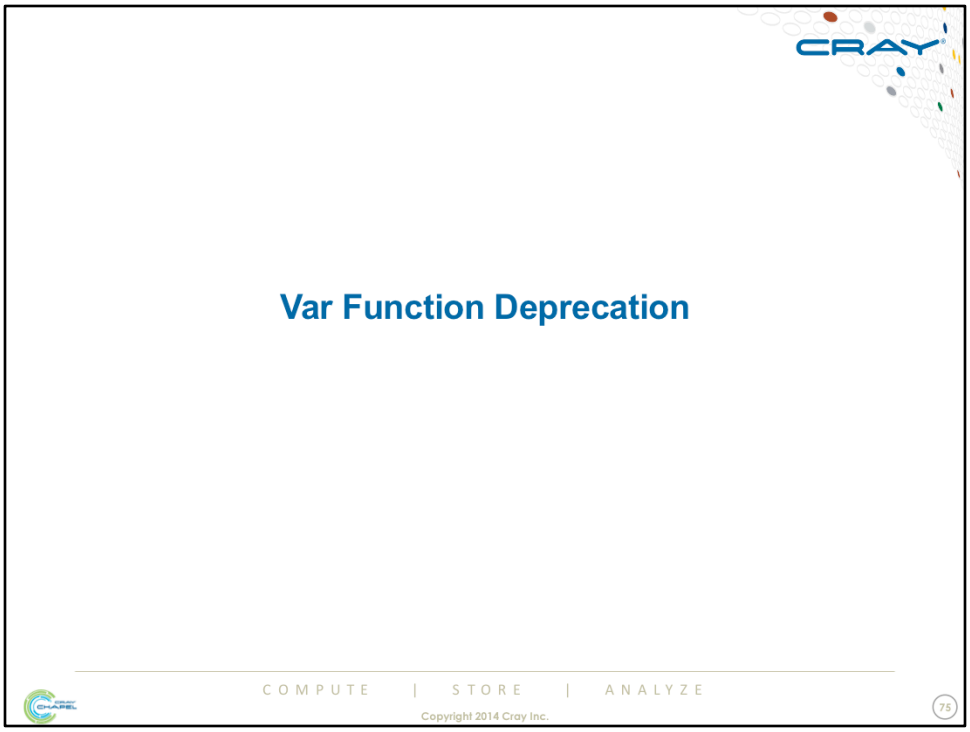
  - previous syntax is supported until v1.11
    - if used, compiler prints a warning

Task Intent Syntax: Next Steps

- **Extend task intents to forall loops**
  - for consistency with tasks created by **coforall** etc.
  - to prevent similar race conditions
- **Add support for other intents**
  - **in** intent → task-private variables
  - **reduce** intent → reduce task-private variables at task exit
    - should support implementing reductions in modules rather than compiler

```
var A: [D] real;
var sum: real;
forall a in A with (+ reduce sum) do
  sum += a;
writeln("The sum of A's elements is ", sum);
```

task-private copies of sum *are added up when loop completes*


Var Function Deprecation
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
75

Var Function Deprecation

Background: Var functions have been a source of confusion

- Marking a function with var indicated that it returned an l-value

```
proc getFoo() var { ... }
getFoo() = 3;
```

- This was implemented before 'ref' was a keyword
- We now use 'ref' in several other places to indicate similar things

This Effort: 'var' has been deprecated in favor of 'ref'

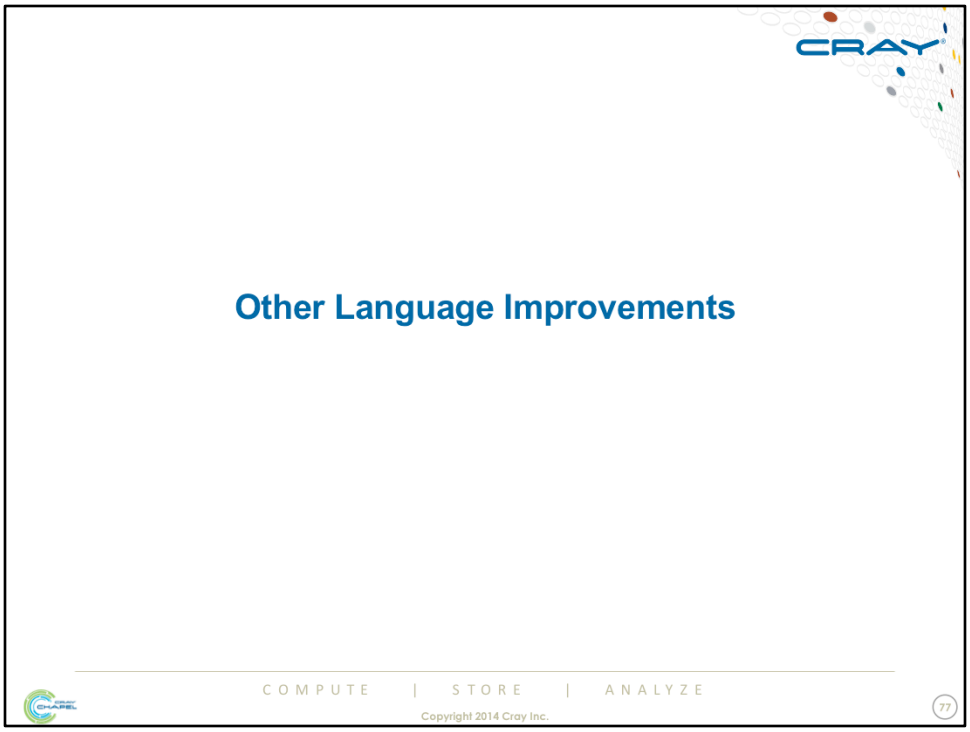
```
proc getFoo() ref { ... }
```

Impact: Consistent use of 'ref' to indicate passing by reference

```
proc ref ClassBar.baz(ref x) ref { ... }
```

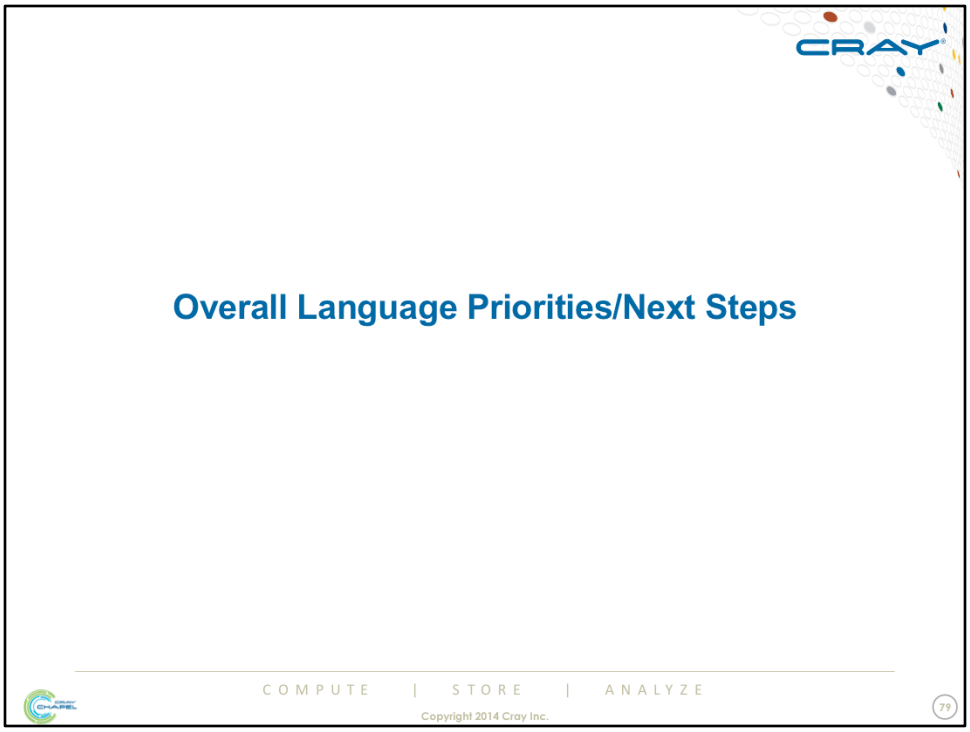
Next Steps: Remove 'var' in the next release

- A warning is displayed if 'var' is used now


Other Language Improvements
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
77

Other Language Improvements

- **Support for octal literals**
  - e.g., 0o777
- **logical negation of integers**
  - e.g., `if (!count) { … }` is now supported for `count: int;`


Overall Language Priorities/Next Steps
COMPUTE | STORE | ANALYZE
Copyright 2014 Cray Inc.
79

Overall Language Priorities/Next Steps

- **Clean-up:**
  - extend sync/single argument passing to other intents
  - fix zippered iteration/promotion for associative domains/arrays
  - remove warnings for deprecated syntax
  - rename ‘refvar’ to ‘ref’
- **Work items:**
  - task intent improvements
    - support for data parallelism
    - support for intents other than ‘ref’
    - enable re-implementation of reductions as module-level code
  - string improvements
    - change default ‘string’ to use record implementation
    - make strings UTF-8
  - refine and implement constructor/default init/noinit story
  - pass contiguous multidimensional arrays to extern procedures
  - efficient vector array reallocation

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, URIKA, and YARCDATA. 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
CHAPEL
CRAY
THE SUPERCOMPUTER COMPANY
http://chapel.cray.com
chapel_info@cray.com
http://sourceforge.net/projects/chapel/