Language and Compiler Improvements

Chapel Team, Cray Inc.
Chapel version 1.14
October 6, 2016
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.
Outline

- Reduction Improvements
- Passing Chapel Functions to Extern C Routines
- Error Message Improvements for Const Intents
- Trailing Commas in Tuple and Array Literals
- Initializers (work-in-progress)
- Recursive Initializers
- Range and Domain Casts
- Requiring Qualified Module Symbol Accesses
- No ‘auto-use’ in nested modules
- enum.size / enum-erators
- Type Iteration
- Casting between real and imag
- Type Queries for First Class Functions
- ‘param’ Improvements
- Improved compilerError(), compilerWarning(), compilerAssert() Signatures
- Other Language Improvements
Reduction Improvements

Four Areas of Improvements for Reductions:

- "reduce=\" Operator For Reduce Intents
- Support Distinct Input/State/Output Types for Reduce Intents
- Reduce Intents with Arrays
- Improve Implementation of Reductions of Forall Expressions
"reduce=" Operator For Reduce Intents
Reduce intents currently require users to have knowledge of the accumulation operation:

```plaintext
var x: InterestingResult;
forall a in A with (InterestingOp reduce x) {
    ....
    x = oneInterestingFunction(3*a, x);
}
writeln("InterestingOp reduction produced: ", x);
```

- **must know type of result**
- **must know how to accumulate input**
- **can use third-party / library reduction operators**
reduce= : This Effort

Introduced a **reduce=** operator
accumulates values into reduce-intent variables

```plaintext
var x: InterestingResult;
forall a in A with (InterestingOp reduce x) {
    ....
    x reduce= 3*a;
}
writeln("InterestingOp reduction produced: ", x);
```

no need to figure out how to write
accumulation operation for InterestingOp

"accumulate 3*a into x"
reduce= : Status

- reduce= available for standard reductions except minloc/maxloc:
  + * min max && || & | ^
  - challenge for minloc/maxloc relates to their use of zippering

- user-defined reductions must implement a method to enable reduce=
  proc accumulateOntoState(ref state, input) { ... }
  - see a simple example here: http://chapel.cray.com/docs/1.14/technotes/reduceIntents.html
  - the interface for user-defined reductions is expected to evolve further
reduce= : Next Steps

- **Finalize interface for user-defined reductions**
  - do they provide storage for accumulation state?
  - do they provide locking?
  - allow records instead of / in addition to classes?
    - to eliminate malloc/free overhead

- **Consider adding support for reduction types (?)**
  - further simplify use of third-party reductions, e.g.:

  ```plaintext
  use ReductionLibrary;
  var reductionVar: reduce InterestingReductionOp;
  forall ... {
    some computations;
    reductionVar reduce= currentInput();
  }
  writeln(reductionVar);
  ```

  "reduce" type: the only place for user to know details of the desired reduction

  can omit with-clause for reductionVar

  inside forall, reductionVar stores accumulation state

  outside forall, reductionVar stores reduction result
Support Distinct Input/State/Output Types for Reduce Intents
Reduction Input-State-Output Types: Background

- Reductions are characterized by multiple types:
  - input, accumulation state, output / result
  - e.g. for min-k reduction: if input is $t$, state and output are $k$-ary sets of $t$
  - for many standard reductions (e.g., $+$), these types are all the same

- Using reduce intents required they all be the same type
  - namely, the declared type of the variable specified in the reduce intent

```pascal
var myVariable: MyType;
forall ... with (+ reduce myVariable) {
    myVariable += 1;
}
writeln(myVariable);
```

- reduction implementation assumed input of myVariable's type
- all three were required to be of MyType
- outside forall loop: reduction result (output)
- inside forall loop: accumulation state
- for many standard reductions (e.g., $+$), these types are all the same
Reduction Input-State-Output Types: This Effort

- Allow the three reduction types to differ for reduce intents
  - currently, user must specify input type explicitly in reduce intent

```plaintext
var reduceVar: k*real;
forall ... with (MinK(real) reduce reduceVar) {
    reduceVar reduce= ...;
}
```

- type of accumulation state is inferred
  - given by identity method in reduction class
  - it is the type of the reduction variable e.g. `reduceVar` inside forall-loop
- type of reduction result is the declared type of the reduction variable
Reduction Input-State-Output Types: Next Steps

Next Steps: infer input/state/output types automatically

- yet still permit users to specify if they wish
- challenge: what if there is a circular dependence between types?

```c
forall ... with (MyReduceOp reduce reduceVar) {
    reduceVar reduce = f(reduceVar);
}
```

- possible solution: disallow such cases
Reduce Intents with Arrays
Array Reduce Intents: Background

● Would like to compute a histogram using reduce intents:

```javascript
var histoArray: [1..numBuckets] int;
forall ... with (+ reduce histoArray) do
    histoArray[computeBucketNum(...)] += 1;
showHistogram(histoArray);
```

iterate over data to plot
for each input, increment corresponding bucket

● The above code did not compile
  ● could not use reduce intents on arrays with any standard reduction
Array Reduce Intents: This Effort and Status

This Effort: improved the compiler and modules

Impact: array reduce intents work with:

\[ + \quad * \quad & \quad | \quad ^ \quad \text{user-defined reductions} \]

Status:

- array reduce intents do not work with:
  \[ && \quad || \quad \text{min} \quad \text{max} \]
- these ops do not produce a single boolean for arrays:
  \[ \text{minloc} \quad \text{maxloc} \]
- zippered reduce intents are not currently supported

Next Steps:

- address limitations above
- eliminate memory leaks generated in some cases:
  \[ * \quad | \quad ^ \quad \text{user-defined reductions} \]
Improve Implementation of Reductions of Forall Expressions
Reductions of Forall Expressions: Background

● “Old-style” implementation of reduce expressions
  - expressions like this one
    \[ \text{reduce} (u \times v) \]
  - had been used for all reduce expressions since \sim 2008
  - based on direct calls to leader/follower or serial iterators

● “New-style” implementation of reduce expressions
  - introduced in 1.13
  - based on forall loops with reduce intents
  - provided performance improvements
  - only applied to some reduce expressions
    - e.g. reductions over forall expressions still used “old style”
      - reductions like this still used “old style”
    \[ \text{min} \text{reduce} \ [\text{indx in MatElems}] \text{calcDtHydroTmp} (\text{indx}) \]
Reductions of Forall Expressions: This Effort

This Effort: applied “new-style” implementation to most reductions of forall expressions

\[ \text{min \ reduce } [\text{indx in MatElems}] \text{ calcDtHydroTmp}(\text{indx}) \]

- exceptions: zippered forall expressions; those with filtering predicates
Reductions of Forall Expressions: Impact

**Impact:** significant performance improvements
- 16-locale dense Lulesh: ~20x
- local NPB CG: ~3x
- local 2D matrix multiply: ~4x

**Next Steps:** improve compiler internal representation
- convert all reductions to forall loops
Passing Chapel Functions to Extern C Routines
Function Pointer Interoperability

Background:

- Chapel has good C interoperability
  - ability to interact with external types, variables, functions, …
  - ability to embed extern blocks of C code in Chapel source
- However, hasn’t traditionally supported passing functions to C
  - requested by users wanting to invoke libraries with callbacks

This Effort:

- Added support for a ‘c_fn_ptr’ type to represent function pointers in C
  - currently, treated completely generically: no argument, return types
  - mapped C function pointers in extern blocks to c_fn_ptr as well
- Chapel functions can be passed to such arguments using c_ptrTo():
  ```chapel
  extern proc bar(fn: c_fn_ptr);
  proc foo(x: c_int): c_int { ... }
  bar(c_ptrTo(foo));
  ```
  - note: only makes sense for non-generic, non-overloaded functions
- Documented at: [http://chapel.cray.com/docs/latest/technotes/extern.html#c-fn-ptr](http://chapel.cray.com/docs/latest/technotes/extern.html#c-fn-ptr)
Function Pointer Interoperability

Impact:
- Has improved users’ ability to interoperate with existing C code
  - e.g., can call GSL functions with callbacks now

Next Steps:
- Extend c_fn_ptr type to include argument / return types
  - and typecheck actual arguments to make sure they match
Error Message Improvements for Const Intents
Error Messages for Const Intents

Background:
● Default task and forall intents result in “const” shadow variables:

```c
var cnt: int;
cnt += 1;       // OK when outside any forall or task constructs
forall ... do
  cnt += 1;    // error: illegal lvalue in assignment
```

● Error upon write access surprising to some users
  ● if unfamiliar with task/forall intents, unclear how to correct
  ● recently became #1 question in tutorial hands-on sessions

This Effort: Print explanation along with error

```c
forall ... do
  cnt += 1;    // error: illegal lvalue in assignment
// note: The shadow variable 'cnt' is constant due to forall intents
```

Impact: Helps make users aware of task/forall intents
Trailing Commas in Tuple and Array Literals
Trailing Commas in Tuple and Array Literals

Background: trailing commas were not generally allowed
- added by necessity for 1-tuples such as (3,)
- parentheses without trailing comma are used for grouping
  ex. (3) is just the integer 3
- rationale: seemed syntactically sloppy to permit it in other cases
- however, users have requested it based on convenience in Python, C

This Effort: allow trailing comma in tuples and array literals

Impact: can now write...

```javascript
var my3tuple = (a, b+c, d,);
var my2elemArray = [100, 200,];
```

```
my3tuple.size == 3
my2elemArray.size == 2
```
Initializers (work-in-progress)
Initializers: Background

- Chapel’s traditional constructor story was naïve
  - Became increasingly clear as users/developers leaned on OOP more

- Last release: designed initializers for classes
  - The current plan of record is:
    ```chapel
    proc init() {
      ...
      // Phase 1 – fields are initialized in declaration order
      super.init(); // Call to parent initializer separates Phase 1 and 2
      ...
      // Phase 2 – whole object ready to have methods called on it, etc.
    }
    ```
  - Compiler initializes fields not explicitly set in Phase 1
  - Alternative syntax proposals available
  - Further details in CHIP 10 and the 1.13 release notes on constructors
  - Still needed to finalize:
    - Inheritance story for records
    - Generics
    - Copy initializers
Initializers: This Effort

Summary:

- Defined a transition story
- Added initial support of compliant initializers
- Decided on a strategy for copy initializers
- Chose a strategy for generics
Initializers: Transition

- **Defined a transition story**
  - Old-style constructors still supported and used by many types
  - It is an error to define both an initializer and a constructor on a type
    ```
    class Foo {
        ... // fields
        proc Foo(args) { ... } // constructor
        proc init(args) { ... } // initializer
        // No point supporting calls to both for same type when one will be deprecated
        ...
    }
    ```
  - If neither is provided, an old-style default constructor is generated
    - But planning to start generating default initializers soon
Initializers: Implementation Progress

- **Work-in-Progress: Compliant initializers now supported**
  - Complaint ⇒ the compiler won’t catch all error cases yet
  - Compiler now recognizes init() methods as initializers
  - “new” expressions invoke init() when defined
  - super.init() and this.init() calls now work
    - Indicates the separation between Phase 1 and Phase 2 of the body
    - Fixed bugs in overridden method inheritance
    - Calls to parent types with user-defined constructors are errors
    - Calls to parent types which use the default constructor work
Initializers: Implementation Progress

- Implemented some Phase 1 verification that:
  - Fields are initialized in order
    - no duplicates, no rearrangement
  - Method calls don’t occur in certain circumstances
  - Fields from the parent type are not initialized in the child initializer

- Implemented initialization for omitted fields in Phase 1
  - Uses field initialization value, or default for type if not present
  - Known Bug: shouldn’t do this when initializer uses this.init()
    - The initializer being called must initialize the child’s fields
    - Adding field initialization to the caller duplicates the work
    - So don’t add or allow it
Initializers: Copy initializers

- **Have basic strategy for copy initializers**
  - Previous thinking in the constructors section of the [1.13 release notes](#)
  - Moved away from D’s postblit model since last release
    - Appeared to be unnecessarily complex
    - The case it would help (optimization of =) could be handled in other ways
    - Could be added later if desirable
  - Current strategy is to rely on single argument initializers
    - `proc Foo.init(x: Foo) { ... }`
  - Can replace existing uses of autoCopy/initCopy
Initializers: Generic Constructors

● **Generic Constructors, today:**
  ● Today we require constructor arguments named after generic fields:

    ```chapel
class Foo {
  param a = 1;
  var b;
}
proc Foo.Foo(param a, b) {
  // Can't elide or give these args different names
  ...
}
```

● Rationale: assignments not generally supported on params and types
  ● name-based matching was a way to bind that supported common cases
  ● avoided needing to special-case assignments to types/params in generics
  ● however, this mechanism is confusing for new Chapel users
Initializers: Generic Initializers

- **Current plan for Generic Initializers:**
  - Treat generic fields similar to other fields
    - Support initialization in Phase 1, even if type or param
    - If initialization is omitted:
      - compiler initializes using field initializer/type when provided
      - generates an error otherwise
  - Let the developer control the initializer’s signature
    - No required argument per generic field
    - This serves to separate initializer argument names from field names

```plaintext
class Foo {
  var a: int;
  var b: int;
}

proc Foo.init(x) {
  a = x*2;
  b = x + 1;
}
```

Consistent handling of common cases
Initializers: Next Steps

● **Implement remaining checks and todos:**
  ● Initializers that contain control flow need more work
    ● Specifically when they contain field initialization or super.init()/this.init() calls
  ● Detect use of “this” object in some places during Phase 1
  ● Detect multiple .init() calls in single control flow
  ● Phase 2 checks

● **Add support for:**
  ● Noinit
  ● Generic types
  ● Copy initializers

● **Implement compiler-generated default initializers**

● **Update library and built-in types to use initializers**

● **Retire constructors**
Recursive Initializers
Recursive Initializers

**Background:** Recursive functions must declare their return type
- Compiler can’t infer return types of recursive functions
  - This forces users to declare the return type - a common complaint
- But constructors/initializers are forbidden from declaring a return type
  - This made recursive constructors/initializers unwritable
  - Yet, their return type should be obvious to the compiler

**This Effort:** Taught compiler return type for (recursive) initializers
Recursive Initializers

Impact:

- Recursive initializers now resolve properly
- Rewrote binarytrees with new initializers, yielding a performance gain

Next Steps:

- Improve compiler’s ability to resolve recursive function return types
Range and Domain Casts
Assigning stridable ranges/domains to non-stridable ones...

- did not generate a compiler error
- was allowed only if the stride == 1
- required a runtime check to ensure that stride == 1
- halted the program if the runtime check failed

<table>
<thead>
<tr>
<th>stride == 1</th>
<th>stride != 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>var r1: range(stridable=false);</code></td>
<td><code>var r1: range(stridable=false);</code></td>
</tr>
<tr>
<td><code>var r2 = 1..10 by 1;</code></td>
<td><code>var r2 = 1..10 by 2;</code></td>
</tr>
<tr>
<td><code>r1 = r2; // runtime check of r2.stride</code></td>
<td><code>r1 = r2; // halt – stride mismatch</code></td>
</tr>
<tr>
<td><code>var d1: domain(rank=1, stridable=false);</code></td>
<td><code>var d1: domain(rank=1, stridable=false);</code></td>
</tr>
<tr>
<td><code>var d2 = {1..10 by 1};</code></td>
<td><code>var d2 = {1..10 by 2};</code></td>
</tr>
<tr>
<td><code>d1 = d2; // runtime check of d2.stride</code></td>
<td><code>d1 = d2; // halt – stride mismatch</code></td>
</tr>
</tbody>
</table>
Range and Domain Casts: This Effort

● Assigning a stridable to non-stridable is now a compiler error

```cpp
var r1: range(stridable=false);
var r2 = 1..10 by 1;
r1 = r2; // error: type mismatch assigning ranges with different stridable params

var d1: domain(rank=1, stridable=false);
var d2 = {1..10 by 1};
d1 = d2; // error: cannot assign from stridable domains to unstridable w/out a cast
```

● Added explicit casts for ranges and domains

```cpp
var r1: range(stridable=false);
var r2 = 1..10 by 2;
r1 = r2:range(stridable=false); // r1 is `1..10` - the stride was dropped

var d1: domain(rank=1, stridable=false);
var d2 = {1..10 by 2};
d1 = d2: domain(rank=1, stridable=false); // d1 is `1..10` (no stride)
```
Range and Domain Casts: This Effort

● The safeCast methods check the stride at runtime
  ● Allows stride 1 to convert to unstridable
    ```
    var r1: range(stridable=false);
    var r2 = 1..10 by 1;
    r1 = r2.safeCast(range(stridable=false)); // OK, stride == 1
    
    var d1: domain(rank=1, stridable=false);
    var d2 = {1..10 by 1};
    d1 = d2.safeCast(domain(rank=1, stridable=false)); // OK, stride==1
    
    ● Halts if stride is not 1
    var r1: range(stridable=false);
    var r2 = 1..10 by 2;
    r1 = r2.safeCast(range(stridable=false)); // halt, stride != 1
    
    var d1: domain(rank=1, stridable=false);
    var d2 = {1..10 by 2};
    d1 = d2.safeCast(domain(rank=1, stridable=false)); // halt, stride !=1
    ```
Range and Domain Casts: Impact

- Assigning stridable to non-stridable is now a compiler error instead of runtime

- Explicit casts permit strides to be dropped

- To ensure stride is 1 at runtime, use `safeCast()`
Requiring Qualified Module Symbol Accesses
Qualified Module Symbols: This Effort

Background:
- ‘use’ statements make modules’ symbols available
  ```
  module M { var x, y = 0; }
  use M; ...x... ...y...    // ‘use’ of M permits unqualified access to x and y
  ```
- ‘use’d symbols can be filtered by ‘only’ or ‘except’ clauses
  ```
  use M except y;
  ...x... ...M.y...
  // ‘except’ clause requires full qualification to access ‘y’
  ```
- no good way to require full qualification on all of a module’s symbols

This Effort: Support filtering of all module symbols
- ```
  use M only ;
  use M except *;    // Equivalent to, and redundant with, the previous line
  ...M.x... ...M.y...
  // M’s symbols are available, but must be qualified
  ```

Impact: Supports programming in the fully-qualified style
- e.g., may be attractive to Python programmers
Qualified Module Symbols: Next Steps

Next Steps:

- consider further extensions to ‘use’ to completely hide symbols
  - i.e., even qualified access would not be permitted
  - look to Haskell’s ‘import’ for case coverage

- consider requiring qualified access by default (or decide not to)
  - whatever the decision, need to put this perennial question to rest soon
    + more familiar to Python programmers
    + arguably encourages a more disciplined coding style
      - a big change from traditional Chapel
      - arguably more burdensome
        (without obvious safety benefits in a statically typed language?)

- topic seems related to “sketch” vs. “production” coding modes
No ‘auto-use’ in nested modules
Nested auto-‘use’: Background

Background:

- certain standard modules are automatically ‘use’d by user modules
  - e.g., IO, Math, Assert, Types
  - supports trivial access to common functions and symbols:

```cpp
writeln(“Hello, world!”);
const x = cos(pi / 8), y = sin(pi / 4), small = min(x, y);
assert(y != 0);
```
Nested auto-`use`: More Background

More Background:
- historically, this has been done for all user modules
- this led to understandable confusion in some cases:

```plaintext
module M {
    // automatic 'use' of IO, Math, Assert, Types here...
    var min = 3;       // declare a variable named min
    module Minner {
        // automatic 'use' of IO, Math, Assert, Types here...
        ...min...       // attempt to refer to outer variable fails; refers to min() instead
    }
}
```
Nested auto-‘use’: This Effort

**This Effort:** Only auto-‘use’ modules for top-level user modules

**Impact:**
- Reduces confusion
- Preserves auto-‘use’ed symbols when not shadowed

```javascript
module M {
    // automatic ‘use’ of IO, Math, Assert, Types here...
    var min = 3;   // declare a variable named min
    module Minner {
        // no automatic ‘use’ of IO, Math, Assert, Types here...
        ...min... // attempt to refer to outer variable now works
        ...max(x,y)... // can still refer to other auto-use’d symbols
    }
}
```
Nested auto-‘use’: Next Steps

**Next Steps:** Give users more control over what is auto-use’d?
- provide a hook to avoid auto-‘use’ing modules?
- provide a means to specify a different set of auto-‘use’ed modules?
enum.size / enum-erators
Enum Improvements: Background

**Background:**

- Chapel’s enums support nice conveniences:

  ```chapel
  enum color {red, green, blue};
  var c = "red": color;  // can cast strings to/from enums
  writeln(c);           // can print enums
  ```

- However, other aspects remained clumsy:
  - No way to know number of enum symbols led to using C-style workaround:
    ```chapel
    enum color {red=0, green, blue, numColors};
    var A: [1..color.numColors] color = ...;
    ```
  
  - No way to iterate over enums directly led to lame workarounds:
    ```chapel
    const colors = [color.red, color.green, color.blue];
    for c in colors do...
    ```
Enum Improvements: This Effort

This Effort: Add new enum features to improve these areas:

```javascript
enum color = {red, green, blue};
var A: [1..color.size] color = ...;   // query an enum’s size
for c in color do...                // support iteration over enums
```

Impact: Permitted us to clean up some shootout benchmarks
- **chameneosredux**: was using workaround to iterate over enums
- **meteor**: was using C trick to compute using an enum’s size

Next Steps: Keep an eye out for other similar improvements
Type Iteration
Type Iteration: Background and This Effort

**Background:** Chapel has not supported iteration over types
- haven’t had strong motivation to do so, nor reasons to disallow
- desire to iterate over enums was a first motivating case
- why not let general types support iteration as well?
  - type iterator methods are a natural mechanism for doing so:
    ```
    record R {
      iter these() { ... } // supports iteration over objects of type ‘R’
      iter type these() { ... } // supports iteration over ‘R’ itself
    }
    ```

**This Effort:** Add support for serial iteration over types
- primary challenge: zippered iteration
  - past implementation of zippered iteration creates tuples of the iterands
  - but Chapel doesn’t support mixing types and values in an enum
  - required rewriting implementation to avoid tuples
Type Iteration: Impact and Next Steps

**Impact:**
- provides new user capability
- supported enum-erator work
  - permits zippering enums with other values
    
    ```python
    for (c, i) in zip(color, 1..) do ...
    ```

**Next Steps:**
- look for other cases that would benefit from type iteration
- extend type iteration to support parallel iteration
  - requires rewriting more zippered iteration code to avoid tuples
Casting between real and imag
Casting between real and imag

**Background:** casts between `real` and `imag` returned 0:
- mathematical rationale: promote to complex and drop a component
  ```
  writeln(1.0:imag); // output 0.0i
  writeln(2.0i:real); // output 0.0
  ```
- in practice, this is rarely useful, often surprising

**This Effort:** Now such casts preserve the floating point value
  ```
  writeln(1.0:imag); // outputs 1.0i
  writeln(2.0i:real); // outputs 2.0
  ```

**Impact:** Resolves a common stumbling block for users
- simplifies generic code working with `imag`

**Next Steps:** Remove workarounds in module code
Type Queries for First Class Functions
First-Class Function Queries: Background

- No way to reflect about First-Class Function signatures
  - Thus, developers had to pass type information explicitly

- As an example:

```cpp
proc generateArray(n: int, generatingFunction, type retType) {
    var A: [1..n] retType;

    for i in 1..n {
        A[i] = generatingFunction(i);
    }

    return A;
}
```
First-Class Function Queries: This Effort

● Implement two new methods for First-Class Functions:

\[\text{proc } \text{func.argTypes: (type formalType \ldots k)}\]
- returns a tuple with the type of each formal

\[\text{proc } \text{func.retType: type}\]
- returns the type that would be returned if function were invoked

● contributed by Nick Park

● Allows more natural expression of such idioms:

\[\text{proc generateArray(n: int, generatingFunction)}\]{
\[\text{var A: [1..n] generatingFunction.retType;}\]

\[\text{for i in 1..n} \{\]
\[\quad \text{A[i] = generatingFunction(i);}\]
\[\}
\]

\[\text{return A;}\]
}
‘param’ Improvements
**Param Formals of Parallel Iterators**

**Background:** Could not use parallel iterators with param formals

```plaintext
iter myParallelIter(param x, param tag)
    where tag == iterKind.standalone
{ ... }
iter myParallelIter(param x)
{ ... }

forall idx in myParallelIter(5)
{ ... }
```

**This Effort:** Such use is now allowed

**Impact:** Supports partial reductions W.I.P.
Param Formals within Forall Loops

**Background:** Param formals were not recognized as “param” inside forall loops

```
proc test(param x) {
    forall ... {
        param y = x;
    }
}
```

**This Effort:** Now recognized as “param”

**Impact:** Supports partial reductions W.I.P.
Casting “param” Values to Arrays

**Background:** Could not cast integer literals to array types with smaller-sized elements

```plaintext
var A: [1..n] int(8);
var B = 0: A.type;
```

**error:** type mismatch in assignment from int(64) to int(8)

**This Effort:** Can cast now
- if integer literal can be cast to array element type

**Impact:** Can compute multiply, bitwise-or, bitwise-xor reductions on arrays with non-int(64) element types

OK!
Improved `compilerError()`, `compilerWarning()`, `compilerAssert()` Signatures
Improved *compiler*() Signatures

**Background:** Had poor signatures of *compilerError()*, *compilerWarning()*, *compilerAssert()*

```plaintext
proc compilerError(param x:c_string ...?n)
proc compilerAssert(param test: bool, param arg1, param arg2, param arg3)
```

**This Effort:** Up-to-date signatures

```plaintext
proc compilerError(param msg: string ...?n)
proc compilerAssert(param test: bool, param msg: string ...?n)
```

**Impact:** Nicer user documentation matches signatures
Other Language Improvements
Other Language Improvements

● Added support for an ‘align’ operator on domains
  ● the logical extension of supporting ‘align’ on ranges

● Added support for casts from ‘string’ to ‘c[_void]_ptr’
  ● contributed by Nick Park

● Promoted array assignments are now always parallel
  ● previously, we had been arbitrarily serializing non-rectangular cases

● Added support for an optional ‘do’ on ‘otherwise’ clauses
  ● not syntactically necessary, but seemed wrong to prevent it
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.