1 Constructors Proposal

This document proposes a few modifications to Chapel's syntax and implementation to better support its object model and to improve its execution efficiency. These modifications take into account last year's review of the User-Managed Memory (UMM) model, and recent additions to the language to support reference variables.

To help with planning, relative priorities and estimated difficulties are associated with the recommended changes.

1.1 Introduction

The original design for Chapel anticipated that memory management would be handled by garbage collection. However, efficient distributed-memory garbage collection algorithms have not been forthcoming. In the mean time, the desire to support a growing number of users by keeping memory leakage within practical bounds has forced the implementation to provide some means for reclaiming memory.

Since memory reclamation was not part of the original design, its implementation is somewhat ad hoc. A review of the memory management in the Chapel compiler revealed that most of the necessary elements are present in some form. But since these have not been applied uniformly across all types, memory is still leaked.

To provide the user full control over memory management, the compiler needs to be modified to call UMM support routines at all the necessary places in the code. These consist of two particular constructors (the zero-args constructor and the copy-constructor), assignment and the destructor. It is hoped that by applying these calls in a consistent manner, type-specific handling of memory management can be removed and the compiler rendered simpler.

Since the Chapel implementation of user-managed memory was added after the initial design, the specification needs to be updated to integrate this concept with the descriptions of object creation, copying and destruction. This proposal restates the object model as it relates to the current Chapel implementation, and makes recommendations on how the existing specification should be updated. In addition, it provides recommendations for syntactical and semantical modifications that may make that model more usable.

The remainder of this document is organized as follows: Section §1.2 discusses the object life-cycle and provides a brief overview of the User-Managed Memory model. It also contains general information common to the other sections. Section §1.3 presents the proposed syntax and semantics for variable and field declarations outside the context of a constructor definition, highlighting how this differs from the current implementation. Section §1.4 presents the proposed syntax and semantics for constructors, highlighting how this differs from the current implementation.

Appendices are provided for quick reference. Appendix §A.1 provides a summary of the proposed changes. Appendix §A.2 gives examples including declarations for the fundamental types provided by Chapel.

1.2 The Object Lifecycle Model and UMM

The interaction of language semantics and the User-Managed Memory model can be discussed in terms of state transitions in the object lifecycle. For background, a summary of the object lifecycle is presented in the following subsection. We then discuss the transitions made by the current constructor model, and how the proposed constructor model would navigate these same transitions.

1.2.1 The Object Lifecycle

In general, the states an object can be in are:

1. Undefined
2. Uninitialized
3. Default-Initialized
4. Field-Initialized
5. Fully-Initialized (a.k.a. Constructed)
6. Destroyed
7. Reclaimed

Before an object has been allocated, it is in the "undefined" state. This is equivalent to representing its storage with a null pointer; the object *per se* does not exist.

Once memory sufficient to represent the object has been allocated, the object moves to the "uninitialized" state. The name of the object (or the class variable) refers to actual memory, but that memory is in an unknown state.

Default-initialization moves the object to the "default-initialized" state. Each field within an aggregate object is in an initial state consistent with its type, but neither the fields nor the object as a whole have been "constructed". Default-initialization may be applied knowing only the type of the object — or just the types of the fundamental leaves of an aggregate type. In particular, neither the constructors nor the field initializers are consulted in creating a default-initialized object.[1]

Field-initialization consists of bringing each field in an aggregate type into a state consistent with the field declarations. If the field is declared without a specified initializer, then the default constructor (for the type of that field) is called; otherwise, a constructor for the type of that field is called, passing the initialization expression for that field as its operand.

Following field-initialization, the actions specified in the body of the constructor are called. In many cases, the body of the constructor is empty, because the actions of default- and field-initialization bring the object into a state consistent with the desired class invariants. In any case, after the body of one of its constructors has run to completion, the object is fully-initialized (i.e. constructed).

[1] This can be stated more simply using just the restriction against consulting constructors, since field-initialization depends upon construction.

When an object is destroyed, it transitions to the destroyed state. This state is equivalent to "uninitialized", meaning that its contents cannot be relied upon. An implementation may reuse a destroyed object without first returning it to the heap through deallocation.[2]

Once an object has been destroyed, an implementation may reuse or reclaim it. This may happen immediately or at some unspecified future time. After it has been reclaimed, the object returns to the undefined state.

1.2.2 Initialization in the Current Implementation

Initialization is the sequence of actions that take place when a variable is prepared for use within its scope, or when a field of a record or class is prepared for use. Field initialization usually takes place as part of the construction of such object, as outlined above in the object lifecycle description. We discuss the initialization of variables in this section, and treat field-initialization in the following subsection as part of the semantics of construction.

Assuming that type inference and initialization are separable concepts, we would expect there to be three canonical forms for variable declaration and initialization:

```
// type T = ... ;
   var a:T;
   var b:T = noinit;
   var c:T = new T( ... );
```

In the final form, any expression coercible to type `T` should exhibit the same semantics, so we could e.g. have said `var c:T = a;` instead, for example.

In only the final form, the explicit type specifier on the variable may be omitted. In that case the compiler infers the type from the initializer expression. In the other forms, the type specifier is required.[3]

As things stand, type inference in the compiler allows it to convert the expression

```
var c = new T( ... );
```

into its canonical form `var c:T = new T( ... );`. Note that since the inferred type of c is the same as the type of the initializer expression, the initializer expression is always coercible to the variable's type.

The way in which the compiler initializes variables of record type is the most interesting case. There are other types known to the compiler that should also be taken into consideration in forming a complete description of the current behavior. These are considered separately below.

1.2.2.1 Record Variables

When `T` is a record type, we observe that the following Chapel code

[2] After its destruction, an object may still be accessible through the name or reference used to call the destructor. But since its contents cannot be relied upon, it represents a programming error to attempt to read or write an object in this state.

[3] The compiler might still be able to infer the type of the variable depending on how it is used (as it currently does for the return value temporary), but that would entail a change in the semantics of the language.

```
var s:R;
var t:R = noinit;
var u:R = new R();
var v:R = new R(8, 9, 10.0);
var w = new R();
var x = new R(9, 16, 0.5625);
```

gives rise to the corresponding AST:

```
1  var s:R
2  {
3    var type_tmp:R
4    ('move' type_tmp (call _construct_R))
5    ('move' s type_tmp)
6  }
7  var t:R
8  var u:R
9  {
10   var type_tmp:R
11   ('move' type_tmp (call _construct_R))
12   var call_tmp:R
13   ('move' call_tmp (call _construct_R))
14   (call = type_tmp call_tmp)
15   ('move' u type_tmp)
16 }
17 var v:R
18 {
19   var type_tmp:R
20   ('move' type_tmp (call _construct_R))
21   var call_tmp:R
22   var coerce_tmp:uint(64)
23   ('move' coerce_tmp (call _cast 8))
24   ('move' call_tmp (call _construct_R coerce_tmp 9 10.0))
25   (call = type_tmp call_tmp)
26   ('move' v type_tmp)
27 }
28 var w:R
29 ('move' w (call _construct_R))
30 var x:R
31 var coerce_tmp:uint(64)
32 ('move' coerce_tmp (call _cast 9))
33 ('move' x (call _construct_R coerce_tmp 16 0.5625))
```

being roughly equivalent to the following canonical expansion:

```
pragma "no init" var type_tmp:R; type_tmp <- new R();
pragma "no init" var s:R; s <- type_tmp;

pragma "no init" var t:R;

pragma "no init" var type_tmp:R; type_temp <- new R();
pragma "no init" var call_tmp:R; call_temp <- new R();
type_tmp = call_tmp;
pragma "no init" var u:R; u <- type_tmp;

pragma "no init" var type_tmp2:R; type_temp2 <- new R();
pragma "no init" var call_tmp2:R; call_temp2 <- new R(8, 9, 10.0);
type_tmp2 = call_tmp2;
pragma "no init" var v:R; v <- type_tmp2;

pragma "no init" var w:R; w <- new R();

pragma "no init" var x:R; x <- new R(9, 16, 0.5625);
```

where the invented pragma `"no init"` performs basically the same function as the `=noinit` initializer — inhibiting any kind of initialization from being applied to the raw memory representing the object. The invented syntax `<-` expresses the action of the `'move'` primitive, which is to perform a bit-wise copy from the RHS object into the LHS object. This is in contrast to `=`, which calls the assignment operator (i.e. `proc =`) compatible with the types of the two arguments.[4]

It can be observed that the form with the `=noinit` initializer performs no initialization whatsoever. Default-initialization, field initialization and construction are all inhibited. Because construction is not performed, this breaks encapsulation of the corresponding class or record. And because the choice of whether construction is done or not is made at the point of declaration (rather than as part of the class design), the class designer is under no obligation to make any operation on an uninitialized object of that type work correctly. In particular, assignment into an uninitialized object is not expected to work correctly which leaves no pathway for an uninitialized object to ever become properly initialized. A redesign of the "noinit" feature to address these problems is currently underway.

Somewhat to my surprise, the forms involving type inference are not merely normalized to their corresponding canonical forms. They follow a completely different translation pathway. I suspect this is due more to implementation history rather than being a matter of design. In many places, the current implementation requires the construction of a full-up object where the construction of a mere type would suffice.

In terms of user intent, both forms of initialized variable declaration (with and without an explicit type specifier) should have implementations resembling the form that does not contain an explicit type; it is hard to imagine the utility of fully-establishing the type-default value in a variable that is then immediately overwritten.

There is further complexity surrounding whether a record type defines a constructor that can be called with no arguments. In the example code above, it does. So the initial value of a `type_temp` variable is determined by calling that user-defined no-argument constructor. If such a user-defined constructor does not exist, the compiler instead inserts a call to `_defaultOf(t)`, where `t` is the type of that record. The compiler-supplied version of `_defaultOf()` calls the all-fields constructor for the passed-in type, passing no arguments.

This is the same behavior as would be achieved by consistently generating a call to the constructor for that type, passing no arguments. In the case that such a user-defined constructor is supplied, the compiler would preferentially bind that one. In the case that no user-defined constructor that can bind to an empty argument list is supplied, the call would instead bind to the compiler-supplied all-fields constructor.

In the current implementation, it is possible to override the compiler-supplied version of `_defaultOf()`. In that case, the user-defined version of `_defaultOf()` is used in preference to the compiler-provided version. However, it is still the case that `_defaultOf()` will only be called if there is no user-defined constructor for that type that can bind to an empty argument list. Part of this proposal is to change that behavior, as detailed here:

Because `_defaultOf()` produces an object that is properly initialized, it is equivalent in its effect to construction. It should therefore be under the control of the author of that type. Both `_defaultOf()` and the no-argument constructor for a given type have the meaning "give me a default-initialized object of type T", so at least in concept they are aliases for one another. The two differences that exist in the current implementation are that the user-defined no-argument constructor will take precedence over `_defaultOf()` and that `_defaultOf()` bypasses any user-defined no-argument constructor (which doesn't exist, by definition), and calls the compiler-defined all-fields constructor (passing no arguments) instead.

A way to bring these concepts closer into alignment would be to have the compiler-supplied version of `_defaultOf()` for record types call a constructor of that type by name, passing no arguments. This will bind with the compiler-supplied all-fields constructor (passing no arguments) if no user-defined no-argument constructor is supplied, and will bind to the user-defined version if one is present.

[4] The assignment operator is not necessarily trivial. It may perform reference-counting operations, etc.

1.2.2.2 Class Variables

When initializing a variable of class type, the following forms are possible:

```
var c:C; c = new C(false);
var d:C = new C(true);
var e = new C(false);
var f:C = noinit;
```

This produces the following AST

```
1  var c:C
2  {
3    var type_tmp:C
4    ('move' type_tmp (call _defaultOf))
5    ('move' c type_tmp)
6  }
7  var call_tmp:C
8  ('move' call_tmp (call _construct_C))
9  (call = c call_tmp)
10 var d:C
11 {
12   var type_tmp:C
13   ('move' type_tmp (call _defaultOf))
14   var call_tmp:C
15   ('move' call_tmp (call _construct_C))
16   (call = type_tmp call_tmp)
17   ('move' d type_tmp)
18 }
19 var e:C
20 ('move' e (call _construct_C))
21 var f:C
```

which is roughly equivalent to the following pseudo-Chapel code:

```
pragma "no init" var c:C;
pragma "no init" var type_tmp:C; type_tmp <- _defaultOf(C);
c <- type_tmp;

c = new C(false);

pragma "no init" var d:C;
pragma "no init" var type_tmp:C; type_tmp <- _defaultOf(C);
pragma "no init" var call_tmp:C; call_tmp <- new C(true);
type_tmp = call_tmp;
d <- type_tmp;

pragma "no init" var e:C; e <- new C(false);
pragma "no init" var f:C;
```

For class variables, assignment is rigidly defined by the compiler to be a pointer copy, so there is no difference between the assignment operator and move primitives for all class types. Therefore, the actions taken for the separate declaration and initialization (for variable `c`) are the same as those taken for the canonical initialization form (for variable `d`). In minute detail, the first form involves one update of `type_tmp` and two updates to `c` while the second form involves two updates of `type_temp` and only one of `d`. This may have performance impacts depending on where the two are stored.[5]

[5]The temporary variable `type_tmp` is expected to be local.

For the noinit case, it should be noted that the value returned by `_defaultOf()` for any class type is `nil`. But when the `=noinit` initializer is present, `nil` is not generated and assigned to the corresponding variable. In spite of that, we find that both module-level and function-level class variables are initialized to `nil` after all. This zero-initialization is added at code generation time to the backend representation of the variable.

1.2.2.3 Distribution Variables

For the Chapel code:

```
var a: Block(rank=2);
var d: Block(rank=2) = noinit;
var b = new Block({2..5, 2..5});
var c: Block(rank=2) = new Block({1..4, 1..4});
```

we obtain:

```
1   var a:Block(2,int(64))
2   {
3     var type_tmp:Block(2,int(64))
4     var call_tmp:Block(2,int(64))
5     ('move' type_tmp (call _defaultOf))
6     ('move' a type_tmp)
7   }
8   var d:Block(2,int(64))
9   var call_tmp:Block(2,int(64))
10  var b:Block(2,int(64))
11  var call_tmp:range(int(64),bounded,false)
12  ('move' call_tmp (call _build_range 2 5))
13  var call_tmp:range(int(64),bounded,false)
14  ('move' call_tmp (call _build_range 2 5))
15  var call_tmp:domain(2,int(64),false)
16  ('move' call_tmp (call chpl__buildDomainExpr call_tmp call_tmp))
17  ('move' b (call _construct_Block call_tmp))
18  var c:Block(2,int(64))
19  {
20    var type_tmp:Block(2,int(64))
21    var call_tmp:Block(2,int(64))
22    ('move' type_tmp (call _defaultOf))
23    var call_tmp:range(int(64),bounded,false)
24    ('move' call_tmp (call _build_range 1 4))
25    var call_tmp:range(int(64),bounded,false)
26    ('move' call_tmp (call _build_range 1 4))
27    var call_tmp:domain(2,int(64),false)
28    ('move' call_tmp (call chpl__buildDomainExpr call_tmp call_tmp))
29    var call_tmp:Block(2,int(64))
30    ('move' call_tmp (call _construct_Block call_tmp))
31    (call = type_tmp call_tmp)
32    ('move' c type_tmp)
33  }
```

It appears that the `call_tmp` variables declared on lines 4, 9 and 21 are never used. Otherwise, this code looks quite similar to initialization for a normal class variable.

1.2.2.4 Array Variables

The Chapel code:

```
var A: [1..n, 1..n] int;
var B = ["1", "2", "3", "4", "5"];
var C: [1..n] real = [.500, .666, .750, .800];
var D: [1..n, 1..n] int = noinit;
```

produces the following AST:

```
var A:[domain(2,int(64),false)] int(64)
{
  var type_tmp:[domain(2,int(64),false)] int(64)
  var call_tmp:domain(2,int(64),false)
  ('move' call_tmp (call chpl__ensureDomainExpr 1..n 1..n))
  var call_tmp:_RuntimeTypeInfo
  ('move' call_tmp (call chpl__buildArrayRuntimeType call_tmp))
  var _runtime_type_tmp_:domain(2,int(64),false)
  ('move' _runtime_type_tmp_ ('.v' call_tmp dom))
  var _runtime_type_tmp_:[domain(2,int(64),false)] int(64)
  ('move' _runtime_type_tmp_ (call chpl__convertRuntimeTypeToValue _runtime_type_tmp_))
  ('move' type_tmp (call chpl__autoCopy _runtime_type_tmp_))
  ('move' A type_tmp)
}
var B:[domain(1,int(64),false)] string
var call_tmp:[domain(1,int(64),false)] string
('move' call_tmp (call chpl__buildArrayExpr "1" "2" "3" "4" "5"))
('move' B (call chpl__initCopy call_tmp))
var C:[domain(1,int(64),false)] real(64)
{
  var type_tmp:[domain(1,int(64),false)] real(64)
  var call_tmp:domain(1,int(64),false)
  ('move' call_tmp (call chpl__ensureDomainExpr 1..n))
  var call_tmp:_RuntimeTypeInfo
  ('move' call_tmp (call chpl__buildArrayRuntimeType call_tmp))
  var _runtime_type_tmp_:domain(1,int(64),false)
  ('move' _runtime_type_tmp_ ('.v' call_tmp dom))
  var _runtime_type_tmp_:[domain(1,int(64),false)] real(64)
  ('move' _runtime_type_tmp_ (call chpl__convertRuntimeTypeToValue _runtime_type_tmp_))
  ('move' type_tmp (call chpl__autoCopy _runtime_type_tmp_))
  var call_tmp:[domain(1,int(64),false)] real(64)
  ('move' call_tmp (call chpl__buildArrayExpr 0.5 0.666 0.75 0.8))
  (call = type_tmp call_tmp)
  ('move' C type_tmp)
}
var D:[domain(2,int(64),false)] int(64)
var call_tmp:domain(2,int(64),false)
('move' call_tmp (call chpl__ensureDomainExpr 1..n 1..n))
var call_tmp:_RuntimeTypeInfo
('move' call_tmp (call chpl__buildArrayRuntimeType call_tmp))
var _runtime_type_tmp_:domain(2,int(64),false)
('move' _runtime_type_tmp_ ('.v' call_tmp dom))
var _runtime_type_tmp_:[domain(2,int(64),false)] int(64)
('move' _runtime_type_tmp_ (call chpl__convertRuntimeTypeToValue _runtime_type_tmp_))
('move' D (call chpl__autoCopy _runtime_type_tmp_))
```

Some nontrivial work goes into initializing an array, so there may not be an obvious translation to the common object construction model. Let us examine the simplest case first — the one involving `B`.

The `call_tmp` is inserted in the normalize pass, but otherwise the AST is passed through unchanged from the form it had immediately after parsing. Resolution must infer the type of the `call_tmp` from the initialization expression, and this is propagated backward to the declared variable as well.

Here is a case where *in situ* initialization does not seem to fit as well as with other types. It appears from the formulation that `chpl__buildArrayExpr()` may return an array of arbitrary shape and size, and so considerable work is done

to build up that array before it is handed off to the copy-constructor (here called `chpl__initCopy`) to be copied into the destination variable. However, that is only a matter of appearance. In fact, the type of the return value of `chpl__buildArrayExpr()` is known well before its execution commences. (It is, in fact, known here by the end of resolution.) Since the type of the resultant object is known, the call to the free function `chpl__buildArrayExpr()` could be replaced by a constructor call, and the call to the array's copy constructor thereby avoided.

The return type of `chpl__buildArrayExpr()` is inferred from the types of the elements in the array literal.[6] The routine itself defines the domain to be a one-dimensional rectangular domain. That is sufficient information to build the domain and array types.

Aside from the obvious performance gains that may be obtained by avoiding an element-by-element copy, there is quite a bit of flexibility lost in first creating an array to express the array literal. An impoverished representation of the initializer list (lacking domain information) could be passed to an array whose type (or at least domain) had already been declared. The same initializer syntax could thus be used to initialize arrays of arbitrary dimension or element type — according to the rules set down by the construct corresponding to that domain type. (I assume that `where` clauses can be used here for partial specialization.)

An alternate approach suggested in the existing spec is to provide some syntax for specifying a domain as part of the literal. This is a workable solution, and only suffers from the fact that a verbatim copy is still required after the full-fledged literal array is built.[7]

Since array support for the `noinit` initializer is not yet provided, the generated code for cases `A` and `D` should be identical. In fact, they differ because the initializer for `A` moves through a `type_tmp` variable and that for `D` does not. The initializers are processed through two different code paths, but they end up in approximately the same place. The paths are expected to diverge when `noinit` is implemented.

The remainder of the code for these two cases first creates a domain expression and then passes this to `chpl__buildArrayRuntimeType`. That function primarily provides the means for grafting the element type onto an existing domain. An array constructor with a generic type argument can perform this grafting equally well. Initialization is not performed when that array object is created. The function as written is split, so the type can be computed first and initialization performed later

The domain is then copied by reference into the array type. This copy may be redundant, depending on how the constructor for the array type is defined. However, it is just a pointer copy after all, so is relatively cheap. Next, `chpl__convertRuntimeTypeToValue` is called on the newly-created array type. That function performs the initialization as originally written in the buildArrayRuntimeType function.

From the results of resolution, it is evident that the type of an array is completely known after resolution. The "runtime type" mechanism may help to manage type information up to a point, but by the time the call to `chpl__convertRuntimeTypeToValue` is called, the type of the array being initialized is fully known. That suggests that the array initialization could be replaced by a constructor call.

The remaining case to consider is the initialization code for `C`. The generated code is the same in every respect with that for `A` and `D`, except that the type-based initialization concludes with the autoCopy call. The array literal is built by the call to `chpl__buildArrayExpr`, and this is followed by assignment to the `type_tmp`. Finally, the `type_tmp` is assigned to the the named variable. It gets the job done, but in a surprising and somewhat roundabout way.

The autoCopy to initialize the `type_tmp` is arguably correct. If that much is done to establish the type of the array, then assignment is the correct choice for updating with values from the initializer. Even so, since the type of the LHS

[6] Actually, only the type of the first element is used in type inference. The elements of the temporary array are assigned in a param loop — each element of the literal list being coerced to match the type of the fist. If there is a mismatch, it is discovered when the body of the loop is resolved.

[7] The literal copy could also be avoided if memory management in the compiler were sophisticated enough to transfer ownership of that array from an unnamed temporary to the named array variable. That capability has been discussed and is feasible, but it makes good sense to first put in place the basics discussed in this document.

is fully known by this point, a constructor that can accept the array literal as an argument can appear instead. The move performs a bitwise copy of the `type_tmp` variable into the named variable `C`. This is also correct — assuming that no attempt is made to reclaim resources initially held by the `type_tmp`. Ownership of them has been transferred to C.

1.2.2.5 Domain Variables

The Chapel code:

```
var D:domain(2);
var E = {1..2, 3..4};
var F:domain(2) = {6..7, 8..9};
var G:domain(2) = noinit;
```

produces the AST:

```
1  var D:domain(2,int(64),false)
2  {
3    var type_tmp:domain(2,int(64),false)
4    var call_tmp:_RuntimeTypeInfo
5    ('move' call_tmp (call chpl__buildDomainRuntimeType defaultDist))
6    var _runtime_type_tmp_:DefaultDist
7    ('move' _runtime_type_tmp_ ('.v' call_tmp d))
8    var _runtime_type_tmp_:domain(2,int(64),false)
9    ('move' _runtime_type_tmp_ (call chpl__convertRuntimeTypeToValue _runtime_type_tmp_))
10   ('move' type_tmp (call chpl__autoCopy _runtime_type_tmp_))
11   ('move' D type_tmp)
12 }
13 var E:domain(2,int(64),false)
14 var call_tmp:domain(2,int(64),false)
15 ('move' call_tmp (call chpl__buildDomainExpr 1..2 3..4))
16 ('move' E (call chpl__initCopy call_tmp))
17 var F:domain(2,int(64),false)
18 {
19   var type_tmp:domain(2,int(64),false)
20   var call_tmp:_RuntimeTypeInfo
21   ('move' call_tmp (call chpl__buildDomainRuntimeType defaultDist))
22   var _runtime_type_tmp_:DefaultDist
23   ('move' _runtime_type_tmp_ ('.v' call_tmp d))
24   var _runtime_type_tmp_:domain(2,int(64),false)
25   ('move' _runtime_type_tmp_ (call chpl__convertRuntimeTypeToValue _runtime_type_tmp_))
26   ('move' type_tmp (call chpl__autoCopy _runtime_type_tmp_))
27   var call_tmp:domain(2,int(64),false)
28   ('move' call_tmp (call chpl__buildDomainExpr 6..7 8..9))
29   (call = type_tmp call_tmp)
30   ('move' F type_tmp)
31 }
32 var G:domain(2,int(64),false)
33 var call_tmp:_RuntimeTypeInfo
34 ('move' call_tmp (call chpl__buildDomainRuntimeType defaultDist))
35 var _runtime_type_tmp_:DefaultDist
36 ('move' _runtime_type_tmp_ ('.v' call_tmp d))
37 var _runtime_type_tmp_:domain(2,int(64),false)
38 ('move' _runtime_type_tmp_ (call chpl__convertRuntimeTypeToValue _runtime_type_tmp_))
39 ('move' G (call chpl__autoCopy _runtime_type_tmp_))
```

As for arrays above, the forms for D and G are nearly the same, except that the form for D uses an extra `type_tmp` that is not present in the form for `G`. This is again due to the different path taken for default initialization as compared to the path taken for the `noinit` initializer. In effect they are the same.

The initialization for `E` is relatively simple, and arguably correct. One could propose an implementation that used a specialized constructor in place of the call to `chpl__buildDomainExpr`, but the code as it is is efficient enough, and perhaps slightly easier to understand.

The form for `F` also echoes the similar case for arrays, and should be approached with the same recommendations: Replacing the creation of a `type_tmp` to represent the type of the domain and then overwriting this with the domain expression only makes sense if the types of the two differ. Otherwise, the initialization can be done in-place and the autoCopy and assignment both avoided. The final copy of the `type_tmp` back into the named variable is also unnecessary, but it basically devolves to a pointer copy and is therefore relatively cheap.

1.2.2.6 Sync and Single Variables

The Chapel code:

```
var a: sync int;
var b: sync int = noinit;
var c: sync int = n;

var d: single int;
var e: single int = noinit;
var f: single int = n;
```

produces:

```
1  sync var a:_syncvar(int(64))
2  {
3    sync var type_tmp:_syncvar(int(64))
4    sync var call_tmp:_syncvar(int(64))
5    ('move' type_tmp (call _defaultOf))
6    ('move' a type_tmp)
7  }
8  sync var b:_syncvar(int(64))
9  sync var call_tmp:_syncvar(int(64))
10 ('move' b (call _defaultOf))
11 sync var c:_syncvar(int(64))
12 {
13   sync var type_tmp:_syncvar(int(64))
14   sync var call_tmp:_syncvar(int(64))
15   ('move' type_tmp (call _defaultOf))
16   (call = type_tmp n)
17   ('move' c type_tmp)
18 }
19 single var d:_singlevar(int(64))
20 {
21   single var type_tmp:_singlevar(int(64))
22   single var call_tmp:_singlevar(int(64))
23   ('move' type_tmp (call _defaultOf))
24   ('move' d type_tmp)
25 }
26 single var e:_singlevar(int(64))
27 single var call_tmp:_singlevar(int(64))
28 ('move' e (call _defaultOf))
29 single var f:_singlevar(int(64))
30 {
31   single var type_tmp:_singlevar(int(64))
32   single var call_tmp:_singlevar(int(64))
33   ('move' type_tmp (call _defaultOf))
34   (call = type_tmp n)
35   ('move' f type_tmp)
36 }
```

Except for the presence of a `type_tmp` for `a`, the cases for `a` and `b` are the same. The case for `c` also contains an initializer that overwrites the initial value supplied by `_defaultOf`. Both forms could be replaced by a suitable constructor call.

The cases for single variables are symmetrical with sync variables, and the same comments apply.

1.2.2.7 Tuple Variables

For the Chapel code:

```
var a:2*real;
var b:2*real = noinit;
var d:2*real = (12.0,13.3);
var e = d;
```

we obtain

```
1  var a:2*real(64)
2  {
3    var type_tmp:2*real(64)
4    var call_tmp:2*real(64)
5    ('move' type_tmp (call _defaultOf))
6    ('move' a type_tmp)
7  }
8  var b:2*real(64)
9  var call_tmp:2*real(64)
10 var d:2*real(64)
11 {
12   var type_tmp:2*real(64)
13   var call_tmp:2*real(64)
14   ('move' type_tmp (call _defaultOf))
15   var call_tmp:2*real(64)
16   ('move' call_tmp (call _build_tuple 12.0 13.3))
17   (call = type_tmp call_tmp)
18   ('move' d type_tmp)
19 }
20 var e:2*real(64)
21 ('move' e (call chpl__initCopy d))
```

It is interesting to note that for tuple initialization, there appears to be an extra `call_tmp` for the cases with an explicit type specifier that is inserted and never actually used in the initialization. Other than that, the tuple code looks quite similar to the code for scalar variables.

1.2.2.8 Scalar Variables

Scalar variables include those of Boolean, integer, real, complex, enumeration and string types. As with other types, the forms of variable declaration are:

```
// type T = ... ;
   var a:T;
   var b:T = noinit;
   var c:T = new T( ... );
   var d = 13; // Initializer expression e has type T.
   var e = d; // Same.
```

The third form is not currently supported for integer or real types, because the syntax type(size)() in a "new" expression is not recognized as a type expression followed by a call (which should invoke a constructor for that type). The above code results in the following AST:

```
1     var a:int(64)
2     {
3       var type_tmp:int(64)
4       ('move' type_tmp 0)
5       ('move' a type_tmp)
6     }
7     var b:int(64)
8     var d:int(64)
9     {
10      var type_tmp:int(64)
11      ('move' type_tmp 0)
12      (call = type_tmp 13)
13      ('move' d type_tmp)
14    }
15    var e:int(64)
16    ('move' e (call chpl__initCopy d))
```

The first form results in default-initialization with a value of the declared type. Note that there is no initialization at all (in the AST) for the form involving `=noinit`. Again, the canonical version contains extra data motion that is undesirable. In particular, it uses assignment, which should not appear in initialization.

1.2.3 Construction in the Current Implementation

In the current implementation, construction involves all of the states from undefined through fully-initialized.

1.2.3.1 Allocation

Variables of class type are only allocated explicitly through the `new` operator. The allocation is performed in the wrapper function which calls the compiler-generated "all-fields" constructor for that class, passing in all default-valued field arguments.

```
1   // This is the AST clipped from the compiler-supplied constructor for a class C.
2   function C._construct_C() : C
3   {
4     var this:C
5     var call_tmp:int(64)
6     ('move' call_tmp ('sizeof' C))
7     var cast_tmp:opaque
8     ('move' cast_tmp (call chpl_here_alloc call_tmp 28))
9     ('move' this ('cast' C cast_tmp))
10    ('setcid' this)
```

Storage for all other types is allocated either statically or on the stack. In particular, variables of record type are not heap-allocated. Therefore, the allocation code inserted in the class constructor wrapper shown above is *not* inserted in the compiler-generated default constructor.

1.2.3.2 Default Initialization

Default-initialization is applied before field-initialization, and this occurs before any assignments that occur within the body of the constructor.

In record and class constructors, this is done on a field-by-field basis, recursively, according to the default value specified for each fundamental type. For the record:

```
record R {
  var u:uint = 7;
  var i      = -2;
  var r:real;
}
```

we have:

```
1     var this:R
2     ('.=' this u 0)
3     ('.=' this i 0)
4     ('.=' this r 0.0)
```

This code appears in every wrapper constructor that is used to call the compiler-generated all-fields constructor, but not in the all-fields constructor itself. The all-fields constructor can be distinguished by the presence of a "meme" argument. This argument does not appear in any of the constructor-wrapper functions that call the all-fields constructor nor in user-supplied constructors.

For a class object of type C containing two records of the above type:

```
class C {
  var uninit:R;
  var init = new R(3, 2, 1.0);
}
```

the default-initialization code looks like:

```
1     var this:C
2     // Allocation code elided. See above.
3     var _init_class_tmp_:R
4     ('.=' _init_class_tmp_ u 0)
5     ('.=' _init_class_tmp_ i 0)
6     ('.=' _init_class_tmp_ r 0.0)
7     ('.=' this uninit _init_class_tmp_)
8     var _init_class_tmp_:R
9     ('.=' _init_class_tmp_ u 0)
10    ('.=' _init_class_tmp_ i 0)
11    ('.=' _init_class_tmp_ r 0.0)
12    ('.=' this init _init_class_tmp_)
```

Note that default-initialization has been recursively inlined. This AST snapshot is taken at the end of the "resolve" pass — well before explicit inlining is applied — so we may conclude that this inlining is performed as part of normalization or resolution.

The default-initialization code is injected by the compiler into every constructor that can be called directly from user code.

1.2.3.3 Field Initialization

Field-initialization in class and record construction is performed by the compiler-generated all-fields constructor. Example AST for this function for a record is:

```
function R._construct_R(arg u:uint(64), arg i:int(64), arg r:real(64), arg meme:R) : R
{
  // Move meme into this.
  var this:R
  ('move' this meme)
  // Perform field-by-field copy.
  ('.=' this u u)
  ('.=' this i i)
  ('.=' this r r)
  // Return this.
  (return this)
}
```

and for a class:

```
function C._construct_C(arg uninit:R, arg init:R, arg meme:C) : C
{
  // Move meme into this.
  var this:C
  ('move' this meme)
  // Construct the base-class sub-object.
  var _tmp:object
  ('move' _tmp ('.v' this super))
  var _return_tmp_:object
  ('move' _return_tmp_ (call _construct_object _tmp))
  // Perform field-by-field copy.
  ('.=' this uninit uninit)
  ('.=' this init init)
  // Return this.
  (return this)
}
```

Both begin by moving `meme` into `this`. The class constructor then calls its base-class constructor, passing in the pointer to the base-class sub-object. Every class inherits from `object`. The sub-object of type `object` contains the class ID and in this way, the run-time type of an object is established. As an optimization, objects whose methods are statically bound can be declared withe the `"no object"` pragma — in which case the `object` base class object is elided.

In the compiler-generated all-fields constructor, it is apparent that the `meme` argument could be eliminated and the initial move avoided if that constructor is called as a method. Aside from dependencies within the compiler on the form of the all-fields constructor function, this change should be relatively small.

1.2.3.4 Construction

In the object lifecycle, construction is the set of actions that take place after field-initialization is complete. This usually corresponds to the body of the constructor function — especially the body of a user-supplied constructor.

In the example code, the user-supplied record constructor looks like:

```
proc R() { r = 2.71; }
```

It sets the r field of that record to a very bad approximation of e.

The corresponding AST at the end of resolution is:

```
function R._construct_R() : R
{
  // The compiler adds this.
  var this:R
  ('move' this (call _construct_R))
  // The original constructor body begins here.
  var call_tmp:_ref(real(64))
  ('move' call_tmp (call r this))
  (call = call_tmp 2.71)
  (return this)
}
```

The compiler post-processes the source code for a constructor in the normalize pass (in `change_method_into_constructor`) to insert a call to the compiler-generated default constructor. For a record, this performs default-initialization followed by field-initialization. Control then advances to the code that constituted the original constructor body.

In the example code, the user-supplied class constructor looks like:

```
proc C(param overw:bool)
{
  if overw then
  {
    uninit = new R(r=3.14);
    init.r = 12.0;
  }
}
```

The interesting case is when `overw` is true. (The param argument causes generic instantiation to stamp out two copies of this constructor — one with the `overw == true` clause and the other without.) The corresponding AST is:

```
function C._construct_C() : C
{
  var this:C
  ('move' this (call _construct_C))
  {
    var call_tmp:_ref(R)
    ('move' call_tmp (call uninit this))
    var call_tmp:R
    ('move' call_tmp (call _construct_R 3.14))
    (call = call_tmp call_tmp)
    var call_tmp:_ref(R)
    ('move' call_tmp (call init this))
    var call_tmp:_ref(real(64))
    ('move' call_tmp (call r call_tmp))
    (call = call_tmp 12.0)
  }
  (return this)
}
```

Note that as with records, the compiler inserts a call to the default constructor for C, moving the result into `this`. The default constructor for C, in addition to performing default- and field-initialization, heap-allocates space to hold the returned object.

1.2.3.5 Post-Construction

A currently-supported feature of the compiler is to insert a call to an `initialize` method at the end of a constructor if such a method is defined in the class.

There are a number of reasons why this language feature should be deprecated, including:

- The same effect can be achieved by calling an `initialize` function explicitly.
- The signature of the built-in `initialize` method is highly constrained (no arguments, no return value) while a user-defined `initialize` method is not so constrained.
- It provides a way for a sophisticated coder to write obfuscated code.
- It slightly complicates the compiler.
- It removes that name from the user's namespace.
- Whatever was the historical problem it solved seems to have been overcome by now.

1.2.3.6 Visibility

The current implementation contains two classes of procedure symbols, *compiler-generated* and *user-defined*. Compiler-generated versions have `FLAG_COMPILER_GENERATED` attached: user-defined versions do not. When a call is being resolved, user-defined versions are considered first. If there are no user-defined candidates, then the compiler-generated versions are examined.

Almost all of the compiler-generated functions, including

- `_defaultOf`
- All-fields constructors
- Record equality and inequality
- `chpl_enum_first`
- Integral-to-enum casts
- c_string-to-enum casts
- Assignment
- Extern init functions
- Record initCopy functions
- Read/write functions

are created with `FLAG_COMPILER_GENERATED` set to true. Certain others, including

- Setters and getters

- `enum_enumerate`
- Record casts (coercions)
- Record hash functions
- enum-to-c_string casts
- Destructors

do not, but special-case code generally allows them to be overridden anyway.

There is an ongoing effort to treat these cases uniformly (i.e. all using the `FLAG_COMPILER_GENERATED` mechanism in `functionResolution`). Irrespective of the implementation details the basic behavior should be that the user can always override the compiler's version if he desires.

1.2.4 Construction in the Proposed Implementation

With the exception of the new initialization syntax proposed in sections §1.3 and §1.4, the observable semantics of construction is expected to be largely compatible with today's implementation.

1.2.4.1 Allocation

Allocation should be moved outside of the definition of a class constructor. Not only will this reduce overall code size, it will allow the structure of class and record constructors to be identical.

1.2.4.2 Default Initialization

The current recommendation is to remove the default-initialization phase as it is currently implemented. Default initialization will be merged with variable or field initialization — meaning that initialization will either be performed in response to an explicit initializer, or through a call to `_defaultOf()`. Since both of these methods would overwrite a zero-initialized variable (or field) what was thought of as default-initialization is now moot. (Refer to the second open issue in §1.4.1 for further discussion.)

As hinted above, an explicit initializer will be used if present, otherwise `_defaultOf()` will be called to supply a default value.

The assignment used to update a typed object with its initializer should be replaced with a copy constructor call.

Constructors should be implemented and called as methods, to avoid the final bit-wise copy-back (and to save confusion over ownership).

When the initializer object is created using a new-expression, the copy-constructor call should be elided (as an optimization).

1.2.4.3 Field Initialization

Field initialization will be handled by the semantics associated with the *initializer−clause*, including substituting explicit or default initializations from the class or record declaration.

The semantic rules for initializations specified in the *initializer−clause* will be consistent with those described for variable declarations.

1.2.4.4 Construction

According to this proposal, the body of a constructor will be "straight code", so no changes are proposed. In particular, no restriction is placed on calling a method of the same class or record from within one of its constructors.

1.2.4.5 Post-Construction

We propose to deprecate and remove the magic surrounding the `initialize()` method.

1.2.4.6 Visibility

Assuming that all of the compiler-generated function classes mentioned above can be overridden in user code, the proposed behavior is the same as the existing behavior, with one exception.

Currently, all-fields constructors remain visible unless they are specifically overridden. We propose that the all-fields constructor for a given type be hidden if the definition of the corresponding type contains one or more explicitly-defined constructors.

The reason for allowing user-code to invoke the all-fields constructor at all is that it supports rapid prototyping. Much of what would be done in an explicit constructor can be simulated in client code. The reason for hiding the all-fields constructor is that it breaks encapsulation. It seems appropriate to hide the compiler-generated all-fields constructor after the first user-defined constructor is present in the code. This means that the class (or record) author is thinking about encapsulation (construction and encapsulation are closely related). At that point, he can add back as much of the all-fields capability as he needs, while appropriately managing the encapsulation of his class (or record).

1.3 Declarations

This section describes the proposed syntax for variable and field declarations, including optional initializers and their impact upon object initialization. This proposal does not address the behavior of the *no−initialization−part* with respect to class and record types. That behavior is the subject of a separate design.

The present proposal seeks to address the sometimes surprising semantics arising from initialization depending on whether the type is specified. It also seeks to define initialization in terms of construction, and remove unnecessary data motion as much as possible.

Thus, the proposal mainly deals with implementation details. The proposed changes should not affect the functional behavior of programs: they should affect only performance. The observable changes will affect the total amount of memory allocated and freed, the amount of processing time spent in data motion and the number of times copy-constructors and copy-assignment operators are called. But filtering out a program's introspection w.r.t. these quantities, its output should remain unaffected.

1.3.1 Syntax

The proposed syntax is unchanged from the existing syntax.

1.3.2 Semantics

The proposed semantic changes are detailed below. The description is in terms of declared variables, but it applies in the same way to fields declared within a class or record.

1.3.2.1 Normalize Type Inference

The current implementation conflates value and type representation to the extent that *representing the declared type of a variable requires creating a default-valued object of that type*. This seems almost self-referential, and probably accounts for much of the complexity surrounding initialization in the compiler, and the extra code emitted when a variable's type is specified explicitly.

At a high level, the proposal here is to perform variable type inference separately from initialization. That is, if a variable (or the return type of a function) has a declared type, that type will be represented internally as a type only. It will not have a value associated with it and it will not entail the creation of a default value of that type as a stand-in for the type itself.

This will require representing the type of a variable separately from its initializer. As it happens, there is already room in the AST for this representation. But the normalization performed in `fix_def_expr()` removes both the `exprType` and `init` fields from a variable declaration and replaces them with flattened AST. The flattened AST creates a temporary object that has the value of the initializer expression (if present) or an invocation of `PRIM_INIT` on the type expression provided. That temporary is copied into the declared variable by assignment.

In contrast, we propose that at least the exprType portion of the representation be left in place, at least until resolution is complete. The expression representing an `exprType` must evaluate to a type. Once type resolution is complete, it can be discarded.

Another observation is that only the type information conveyed by such an expression is important. Any values used in the representation of that expression may be discarded. Unfortunately, it is rather glib to say so. Representation of coercions between fundamental types is currently tightly coupled with their value representation. This seemingly small change in the internal representation will in fact be quite extensive.

1.3.2.2 Unify Initialization

In the current implementation, there is a step where the value is default-initialized prior to its value being established with an initializer if present. If the view is taken that the declaration itself always results in a constructor call (the copy-constructor if an initializer is supplied and the no-args constructor if not), then default initialization becomes moot and should be removed.

As part of this proposal, the existing `_defaultOf()` feature will be preserved. The observation is that a call to `_defaultOf()` is effectively a constructor call. This is backed up by the fact that the compiler-supplied implementation of `_defaultOf()` actually calls the compiler-supplied zero-args constructor.

After the type of a variable is established, it will either have an initialization expression or not. If the initialization expression is missing, the compiler will add a call to `_defaultOf()` to supply the default value corresponding to that type. If the `=noinit` initializer is supplied, then the special actions specified for that initializer are taken. Otherwise, the initializer provided explicitly in the code is used.

Like the `exprType` expression, it may be desirable to leave the initializer attached to the declaration rather than smashing it flat. But in this case, I think either representation will serve: there is no pressing need to un-flatten the representation.

1.3.2.3 Collapse Constructor Calls

In the current implementation, constructor calls allocate and initialize (for class types) or simply initialize (for record types) an unnamed object of the corresponding type. For class types, the resulting object is copied into the named variable using assignment. But since assignment of class variables is rigidly defined by the language to be a bitwise copy of a reference (pointer) to that unnamed object, the copy is not expensive.

For record types, the story is different. In general, the semantics of assignment are not defined if the left operand is in an uninitialized state. So using assignment to initialize the named variable is flat wrong. If the initializer expression is a named object (see below) then at worst copy-construction should be used. But for constructor calls we can do even better by eliminating the copy entirely.

Secondly, it is wasteful of effort to gin up an anonymous record object, copy its contents into a named record object and then throw the original away. It would be better to express a constructor as a method that initializes the memory associated with the named variable in situ. Allocation and deallocation of the unnamed temporary could be avoided, as well as the verbatim copy.

Therefore, the proposal is to represent constructors as methods and the calls to them as method invocations. This will directly support initializing record objects in situ. For symmetry, it is desirable (though less urgent) to pull the allocation part of class object creation (via a `new` expression) outside of the constructor. That would make the internal structure of class and record constructors identical.

Where an explicit initializer or constructor call is not provided, it is similarly desirable to have `_defaultOf()` defined as a method, for the same reasons as described above for constructors. In the current implementation `_defaultOf()` works like a constructor — its default implementation for record types even devolving to a constructor call. Maintaining that symmetry will keep the specification and implementation of `_defaultOf()` simple.

1.3.2.4 Use Copy-Construction, Not Assignment

The cases discussed so far involve no initializer, the `=noinit` initializer and an initializer that looks like a constructor call (i.e. `= new T(...)`. The remaining case is where the initializer expression is an expression coercible to the type of the named variable.[8]

As explained above, assignment in the context of initialization is illegal, because the left operand is not yet initialized. Copy-construction should be used instead. The proposal here is to simply substitute a copy-constructor call where assignment is inserted today. For fundamental and class types, copy-construction, assignment and bit-wise moves are semantically equivalent, so in that realm it becomes merely a name change. The difference between assignment, copy-construction and a bitwise copy is only of consequence as it applies to records and record-based types (e.g. tuples and ranges).

1.3.3 Discussion

Using copy-construction in place of assignment does not currently make any difference as long as the compiler-provided versions of those methods are used. However, for user-defined record types, the assumption of prior initialization of the left operand of assignment can be violated, and such a violation may cause incorrect behavior. I believe we already have one test case that fails (only in multi-locale testing) due to this problem (in privatized arrays, because the default privatization index is not distinguished from a valid one). More such problems may appear as greater use is made of records to support ad hoc memory management schemes. The simple solution is to "do it right".

Normalizing type inference will get rid of the extra type variable initialization code. After which, unifying the initialization code just means adding a `_defaultOf` call wherever no explicit initializer is provided. Collapsing constructor calls will provide some nice code size reduction and performance improvement.

1.3.4 Examples

An example of the odd behavior associated with an explicit type as well as constructor call in an initializer is given by test/types/records/hilde/newUserDefaultCtor.chpl:

```
record R
{
  proc R() { writeln("Called R()."); }
  proc ~R() { writeln("Called ~R()."); }
}

var r:R = new R();

writeln("Done.");
```

The output of this program is:

```
Called R().
Called ~R().
Done.
Called ~R().
```

[8] In the case that the type of the named variable is not specified explicitly, it will be inferred from the computed type of the initializer expression. The inferred type of the variable will of course always match the type of the initializer expression exactly, so the latter is always trivially coercible to the former.

whereas what is expected is:

```
Done.
Called ~R().
```

The difference between calling assignment versus calling the copy-constructor should be made apparent by instrumenting both in a test case.

The other changes — separating out type inference and handling initialization uniformly — can probably only be detected by examining the AST. The supporting change of representing and calling constructors as methods would, for the most part only be visible in the AST. Although (as mentioned above) total memory usage statistics would also be expected to change. The performance benefits of initializing variables in situ might be exposed by calling a function that performs initialization a couple million times and comparing the total elapsed time before and after.

1.4 Constructors

This section describes the proposed syntax and semantics for constructors.

1.4.1 Syntax

A constructor performs initialization of an object. On entry, a constructor may assume that each of its fields has been default-initialized according to the rules for that type, as specified in Section 8.1.1 of the Chapel Language Specification.[9] Default-initialization is supplied by the compiler, so it has no corresponding syntactical element.[10]

> *Open issue*. Taken together, the semantics being proposed for variable and field initialization and construction imply that every such object will be initialized one way or the other. In that case, the default-initialization (a.k.a. zero-initialization) step mentioned in the object lifecycle can be omitted.
>
> I believe that zero-initialization is a part of the C++ specification to support backward compatibility with C, and to allow the creation of global and file-scope static objects that are in a known state before their construction at the start of execution commences. This is important in the execution environment envisioned for programs of that period, especially since the order of initialization of global and file-scope objects from different compilation units is unspecified.
>
> In modern programming practice, however, the use of global objects is strongly discouraged. And if initialization order is strictly specified by the language, then the need for such zero-initialization disappears.
>
> Given its negative impact on performance, the recommendation here is to remove zero-initialization unless a strong case can be made for it. I left it as an open issue in case there are use cases that I have overlooked.

Default-initialization (if performed) is followed by field-initialization. Field-initialization constructs each field in an object in an order specified in the Semantics (§1.4.3 subsection below. The syntactical elements influencing field-initialization are the field declarations and the initializers appearing in the *initializer–list*. Field-initialization is complete before control enters the body of the constructor.

The body of the constructor performs the actual construction of the object. If a class or record type is merely an aggregation of the fields it contains, it imposes no further invariants on its contents. Thus, it is quite common for a constructor body to be empty.

The syntax of a constructor may thus be set down as:

constructor–declaration–statement:
  linkage–specifier$_{opt}$ **ctor** *type–binding constructor–name*$_{opt}$ *argument–list*
    where–clause$_{opt}$ *initializer–clause*$_{opt}$ *constructor–body*

constructor–name:
  identifier

initializer–clause:
  with (*initializer–list*)

initializer–list:

[9] Default-initialization may be overridden for an individual type by defining an explicit `_defaultOf(type t)` for that type.
[10] A separate proposal is being developed for a "no-init" capability, whereby default-initialization may be overridden in whole or in part.

initializer
initializer–list , *initializer*

initializer:
field–name = *expression*

field–name:
identifier

constructor–body:
statement

The linkage specifier (if present) is `inline`, `extern` or `export`. Since in general the backend implementation does not support the object model, the `extern` option probably does not make much sense at present. Constructors are expected to be statically-linked methods, so it should be possible for an external program to call a constructor exported using the `export` linkage-specifier through a stereotyped method interface. The meaning of the `inline` specifier should be self-evident.

The keyword `ctor` introduces a constructor as distinguished from a procedure or iterator. A special keyword was chosen to support the arbitrary naming of constructors – which would be a deviation from the language as currently specified. At present, the name of the constructor must match the type name, so there is no need for a special keyword: the compiler can infer from its name that a function is a constructor.

The *constructor–name* may be omitted, in which case the default constructor name is used internally. The default constructor name is an implementation-defined reserved name that is unique within the (class) scope in which it is defined. If the *constructor–name* is the same as the class name, it is replaced internally by the default constructor name.

> *Rationale*. Programmers familiar with the C++ convention can use the class name to nominate constructors. However, the creation of generic classes is simplified if the constructor name is invariant across classes. Making the constructor name optional serves this purpose by associating the empty string with the same set of overloaded constructors.

Constructor declarations do not have a parentheses-less form.

> *Rationale*. Normal methods may have either a single parentheses-less version or overloaded versions taking various argument lists. In typical usage, we expect overloaded constructors to be common. Also, the parentheses-less form would have the same behavior as the default constructor — that is, it would perform default initialization for all fields. But the compiler-generated constructor already does this (if called with an empty argument list). Given that the parentheses-less form is redundant and potentially takes flexibility away from the user, there is no advantage to supporting it.

If present, the *initializer–clause* contains a comma-separated list of *initializer*s. Each field-name in the initializer list must correspond to the name of one of the fields declared in that class or record, or in one of its base types (recursively). Each initializer contains a field-name followed by an = followed by a value expression for value fields, a param expression for param fields or a type expression for type fields. Each initializer results in the initialization of that field by calling the constructor corresponding to the type of that field, passing in the arguments supplied. In this context, the = does not mean assignment; rather, it implies initialization.

Rationale. Since a new-expression already provides syntax for creating a (class or record) object passing an arbitrary argument list to the constructor, this behavior can be co-opted for the purposes of initialization – thus avoiding a radical syntax change while supporting the required semantics of the added *initializer–clause*.

In terms of implementation, it is understood that initialization of a variable or field using a new-expression does not mean that an anonymous symbol is first built up and then copied verbatim into the named memory location. Rather, the new-expression merely specifies how the named variable is to be constructed in-place.

Within an *initializer–clause*, the name of each field may appear at most once. A field name may contain any number of `super.` qualifiers to distinguish an inherited field from a field of the same name in the most-derived type.

It is illegal to declare a return type for a constructor; the return type is implicitly `void`. This also means that if a `return` statement appears within the *constructor–body*, it must be of the form that does not return an expression.

Within an *initializer–clause*, the names of all formal arguments in that constructor declaration are in scope. All module-level variables may also be referenced. In addition, any field names associated with that constructor's type or any of its base types are in scope. However, if a field name is used in the list of expressions passed as actual arguments to a field-initializer, the field-initializer associated with that name must appear to the left of its use. It is illegal to reference `this` within the *initializer–clause*.

Within the body of a constructor, the names of all formals, field names and visible module-level variables are visible, as well as any variables defined within the body itself.

1.4.2 Binding

In general, a constructor call has the form:

constructor–call–expression:
new *type–name* (*named–expression–list*$_{opt}$)

The *type–name* must name a type. The named type may be generic. The set of visible constructors are all constructors defined for that type. Usual generic instantiation rules are applied to the set of visible functions to select a set of candidate constructors. Any `where` clauses are then applied to reduce the set of candidates. Disambiguation is applied to select the best candidate from among those remaining. It is a program error if there is no unique best candidate. Otherwise, the call is bound to that unique best candidate.

Actual arguments in the call are bound to formal arguments in the constructor according to the usual argument binding rules (including the substitution of default formals for omitted actual arguments).

1.4.3 Semantics

When a constructor is executed, formal initialization occurs first, followed by field initialization and then construction. Formal initialization consists of initializing the formal arguments in the constructor from the passed-in actual arguments using the normal argument-passing rules. Construction consists of executing the body of the constructor. Field initialization is detailed below.

Field initialization starts with the first initializer in the initializer-list and proceeds from left to right. Initialization of the first field in the list completes before initialization of the second field begins and so on. The expressions used in any named-expression passed as an actual argument to a field-initializer call may consist of any expression that is valid in that scope (including function calls and other expressions). In particular, the names of any or all formal arguments of the constructor as well as the names of any fields that have already been initialized may be used. The use of the name of any field that has not yet been initialized is illegal.[11]

After the initializer-list has been processed, some fields may remain uninitialized. These fields are initialized using the initializers specified for them in the class or record declaration. The fields of a class or record are processed in lexical order. If not already initialized, the initializer associated with the field declaration (if any) is used. Otherwise, the field is default-initialized via a call to `_defaultOf()`.

Example (InitializationOrder.chpl).

```
record R {
  var a : int;
  var b : int = 3;  // Same as var b = 3 or var b:int = new int(3);
  var c : int;
  var d = new int(4);
  ctor() with (c = 4, d = c + 1) {}
  ctor(_c) with (a = 2, c = _c, d = a**b) {}
}
```

In this example, the first constructor is legal while the second is not. In the first one, `c` is first initialized to `4`, then `d` is initialized to `5`. It is legal to use `c` in this context because it has already been initialized. Because initializers for `a` and `b` were not listed in the *initializer−clause*, they are initialized as specified by their field declarations in the record itself. In lexical order, `a` appears first. Because no explicit initializer is provided, the `_defaultOf()` method (for a receiver of type `int`) is called on `a`, which has the effect of setting `a` to zero. The declaration for `b` is equivalent to a copy-constructor call for type `int`. It sets the value of `b` to `3`.

The second example would be flagged as containing a coding error. In the initializer list, the field name `b` is accessed in the initialization expression for `d` before `b` has been initialized. This could be repaired by initializing `b` explicitly before the initializer for `d` is encountered. We might consider special syntax, where mentioning just the name of a field would mean "use the field-default initializer for this field. In which case

```
ctor(_c) with (a = 2, c = _c, b, d = a**b) {}
```

would mean, "initialize `a` to `2`, `c` to `_c`, `b` to `3` and then `d` to `8`."

1.4.3.1 `const` Objects and Fields

When a variable or field is declared as `const`, that specifier does not take effect until after its initialization is complete. For a variable, this occurs after the call to the constructor returns. The declaration

```
const r = new R(<args>);
```

is legal, because the above declaration is semantically equivalent to

```
('=' r chpl_mem_alloc(sizeof(R)))
(_construct_R r <args>) // r._construct_R(<args>)
```

[11] This can be reported as a syntax error, since the error condition can be detected statically.

. Since `const`-ness of the object as a whole does not take effect until after the constructor returns, any writable fields in `r` may be overwritten within the body of the constructor.

When a field is declared as `const`, its value is established during initialization. Once processing of the initializer-list within a constructor is complete, it cannot be overwritten. This means that a `const` field cannot be overwritten in the body of a constructor for the type that contains it.

```
record R {
  const _i : int;
  var _r : real = -7.0;

  // Sets _i to the value of the int argument, and _r to 2.0:
  ctor (i : int) with (_i = i) { _r = 2.0; }

  // Error: Cannot write _i in the body of the constructor since it is const.
//  ctor (r : real) with (_r = r) { _i = 3; }
}

const r = new R(4);  // The update of r._r in the body of the constructor is legal.
writeln(r); // (_i = 4, _r = 2.0)
// r._r = 4.2;  // Error: r is const, so no updates after initialization.
```

1.4.4 Discussion

The existing initialization syntax for variable and field initialization can also be employed in the initializer list. Because they specify initialization, the proposed semantics are different from those of the existing implementation. At present, assuming that `R` names a record type

```
var r:R = new R(args,...);
```

is actually implemented by first creating a variable `r` and default-initializing this according to the definition of `R`. It then proceeds to construct an anonymous R, passing in the argument list `args,...`. The value of the resulting anonymous record is then copied into `r` using the assignment operator that can accept an object of type R as it left- and right-hand arguments.

The default-initialization of `r` and its redefinition through assignment are both undesirable. We really want variable and field definitions to construct the corresponding object in-situ. In this view, a variable declared without an initializer should be interpreted as a call to the default (no-args) constructor for that variable's type. Thus

```
var r:R;
```

should be interpreted roughly as

```
r.R();
```

That is, the a constructor for type `R` is called as a method, specifying `r` as the receiver and passing no arguments.[12]

A declaration with an *initializer–part* should be interpreted as a constructor call in which only one argument is passed. If the type of the variable (or field) and the initializer expression are the same (as will always be true if the type of the variable or field is inferred), then the method called will be the copy-constructor for that type. The two types may differ as long as there is a constructor that will accept a single argument of the initialization expression's type. (If not, then an unresolved call error will be reported.)

[12]The `_defaultOf()` feature, as proposed, allows the user to override the compiler-supplied behavior of calling the record's zero-args constructor.

```
var r = s;
```

This says that r is initialized by calling the copy-constructor for the type of `s`. Assuming that `s` is of type `R`, this would be roughly equivalent to

```
r.R(s);
```

As a special case, the syntax

```
var r = new R(args,...);
```

should not construct an anonymous R and call the copy-constructor to then initialize `r`. Rather, it should call the constructor for `R` with the given argument list to initialize `r` in situ. This would be roughly equivalent to:

```
r.R(args,...);
```

It is desirable that the transformation of the two forms involving = into in-place constructor calls take place before resolution. In that way, construction of variables and fields will be called as methods, and all standard steps of resolution (including scope resolution, visible method selection, generic instantiation, candidate selection and argument binding) used in a consistent way.

1.4.5 Examples

As a practical example, we can consider an explicit initializer for one of the array implementation types. At present, the code relies upon the compiler-supplied all-fields constructor and the special `initialize()` function. These two pieces can be combined in an explicit constructor, to make the semantics of initializing an object of that type somewhat more obvious.

The field-declaration part of the that class is given by:

```
class DefaultRectangularArr: BaseArr {
  type eltType;
  param rank : int;
  type idxType;
  param stridable: bool;

  var dom : DefaultRectangularDom(rank=rank, idxType=idxType,
                                  stridable=stridable);
  var off: rank*idxType;
  var blk: rank*idxType;
  var str: rank*chpl__signedType(idxType);
  var origin: idxType;
  var factoredOffs: idxType;
  var data : _ddata(eltType);
  var shiftedData : _ddata(eltType);
  var noinit_data: bool = false;

  ...
}
```

That would make the compiler-generated all-fields constructor look like:

```
class DefaultRectangularArr : BaseArr {
  ctor (type eltType,
        param rank : int,
        type idxType,
        param stridable : bool = false,
        dom = new DefaultRectangularDom(rank=rank, idxType=idxType,
                                        stridable=stridable),
        off : rank*idxType,
        blk : rank*idxType,
        str : rank*chpl__signedType(idxType),
        origin : idxType,
        factoredOffs : idxType,
        data : _ddata(eltType),
        noinit_data : bool = false)
    with (eltType = eltType,
          rank = rank,
          idxType = idxType,
          stridable = stridable,
          dom = dom,
          off = off,
          blk = blk,
          str = str,
          origin = origin,
          factoredOffs = factoredOffs,
          data = data,
          noinit_data = noinit_data) {}
}
```

Open issue. The example is written this way because I realize there is no way to use a qualified name for the constructor but omit the name itself so that the compiler-default constructor name is used. I think the empty string would still work in this context, but it looks a bit awkward to say:

```
ctor DefaultRectangularArr.( ... ) with ( ... ) {}
```

It is worth considering alternatives.

Open issue.

The example code shows that there is a lot of copying of formal arguments in to the *initializer-clause*. As syntactic sugar, we might consider automatically feeding through formals to their like-named fields if that field is not mentioned explicitly in the `initializer-clause`. That would mean that the entire `with` clause could be omitted from the example code above and the overall meaning of the constructor was unchanged.

The `initialize()` routine is called implicitly by each constructor, including compiler-generated ones. The `initialize()` routine for this class is given by:

```
proc initialize() {
  if noinit_data == true then return;
  for param dim in 1..rank {
    off(dim) = dom.dsiDim(dim).alignedLow;
    str(dim) = dom.dsiDim(dim).stride;
  }
  blk(rank) = 1:idxType;
  for param dim in 1..(rank-1) by -1 do
    blk(dim) = blk(dim+1) * dom.dsiDim(dim+1).length;
  computeFactoredOffs();
  var size = blk(1) * dom.dsiDim(1).length;
  data = _ddata_allocate(eltType, size);
  initShiftedData();
}
```

Therefore, our example constructor might look like:

```
ctor DefaultRectangularArr.DefaultRectangularArr(type eltType,
                                                 param rank : int,
                                                 type idxType,
                                                 param stridable : bool = false,
                                                 dom : _domain)
  with (eltType = eltType,
        rank = rank,
        idxType = idxType,
        stridable = stridable,
        dom = dom) {
    if noinit_data == true then return;
    for param dim in 1..rank {
      off(dim) = dom.dsiDim(dim).alignedLow;
      str(dim) = dom.dsiDim(dim).stride;
    }
    blk(rank) = 1:idxType;
    for param dim in 1..(rank-1) by -1 do
      blk(dim) = blk(dim+1) * dom.dsiDim(dim+1).length;
    computeFactoredOffs();
    var size = blk(1) * dom.dsiDim(1).length;
    data = _ddata_allocate(eltType, size);
    initShiftedData();
}
```

Here, we have inlined the body of the existing `initialize` routine in the body of the constructor. More simply, the body of this constructor could be replaced by an explicit call to `initialize()`. According to the present proposal, implicit calls to `initialize()` would be eliminated, so in classes that currently exist for which such a function is defined, it would have to be called explicitly.

The semantics of this constructor are as follows:

1. Formal arguments are initialized by their corresponding actuals.
2. Default values are supplied for unbound formals (e.g. if an actual corresponding to `stridable` is missing, then `false` is used).
3. The initializations specified in the `with` clause are processed left-to-right. The only value field specified in the `with` clause is `dom`; this gets initialized with the domain object that was passed in as an argument.
4. Any value fields left uninitialized at that point are initialized to their default values as specified in the class declaration. Specifically, `off`, `blk`, `str`, `origin`, `factoredOffs`, `data` and `shiftedData` are all initialized to the default value of their respective types. The `noinit_data` field is then initialized to `false`, as specified in the class declaration.
5. Then, the statements in the body of the constructor are executed.

One possible call that would bind to this constructor is taken from the existing implementation of `DefaultRectangularDom.dsiBuildArray(type eltType)`:

```
return new DefaultRectangularArr(eltType=eltType, rank=rank, idxType=idxType,
                                 stridable=stridable, dom=this);
```

1.5 The noinit Initializer

This section describes the proposed syntax and semantics for the `noinit` initializer.

1.5.1 Syntax

The special initializer `noinit` may appear anywhere an *initializer−expression* may appear: in a variable or field declaration or as the initialization expression in an initializer-list. To include `noinit` in the syntax, we replace *expression* with *initializer−expression* in those context and define the latter as:

initializer−expression:
  expression
  noinit

1.5.2 Semantics

When the variable or field being initialized is of a scalar type (integer, real, imaginary, complex, bool, enum), presence of the `noinit` initializer disables the normal default initialization.

When the variable or field being initialized is of class or record type, the `noinit` initializer is internally replaced by a call to the constructor for the type of the destination variable or field, passing the argument "`noinit = true`". Hence the two declarations below are exactly equivalent:

```
var r1:R = noinit;
var r2:R = new R(noinit = true);
```

The `noinit` initializer is essentially syntactical sugar for a specific constructor invocation. Note that when `noinit` is supplied for a variable or field of class type, *more* initialization will be performed than if the class variable had no initializer at all and was thus default-initialized to `nil`. However, that behavior is consistent with the interpretation of the `noinit` initializer as a constructor call.

When the variable or field is of tuple type, the `noinit = true` argument is passed to the constructor for each of its elements.

1.5.3 Discussion

A number of advantages accompany defining the `noinit` feature in this way:

- It is easy to implement.
- It is easy to describe in the specification.
- When explained as a textual substitution, the feature is easy to comprehend.
- The effect of `noinit` is under the control of the record author.

This last point is particularly important, because even when `noinit` is used to initialize a record, the resulting object must be initialized to a point that it obeys the basic invariants pertaining to the assignment operator for that record type.

By using the `noinit` initializer in the initializer-list of a record or class constructor, the sub-objects of a record or class can be left in an uninitialized or partially-initialized state (as defined by that field's type). This permits the effect of the `noinit` flag to be defined recursively. Moreover, it provides a path by which most of the initialization of a record can be inhibited — should the record author deem that desirable.

The example in the next subsection shows that careful coding can result in record initialization that overrides the language default behavior and leaves some or all of the fields partially or completely uninitialized. The way this is achieved under the current proposal is rather verbose, so we might consider alternate syntax that inhibits field initialization by default rather than having it on by default. We might have an alternate keyword to indicate that only the named fields are participate in field-initialization in a constructor. As a strawman, consider `with only`

```
ctor R(x : real)
with only (r = x)
{}
```

In the record type R defined below, this would leave the fields `i` and `u` unitialized and set the value of field `r` to `x`. It then becomes very easy to write a constructor that will perform no initialization at all:

```
ctor R() with only () {}
```

An alternate idea would be to give the noinit constructor its own character. In that case, the `noinit` initializer means "call the noinit constructor passing zero arguments". In the context of a noinit constructor, the with-clause would have a slightly modified meaning: field names not mentioned explicitly would be initialized through a call to their noinit constructors, recursively. Contrast this with the standard with-clause, wherein if a field name is unmentioned it is initialized using that field's class-scope or record-scope initial value.

1.5.4 Examples

For example, in:

```
record R {
  var i: int = -7;
  var r: real = 3.1416;
  var u: uint = 42;

  ctor (param noinit = false)
  with (i = int(noinit=noinit), r = real(noinit=noinit), u = uint(noinit=noinit))
  {}
  ctor (x : real) with (noinit, r = x, noinit) {}
}

var r1:R = noinit;
var r2:R;
var r3:R(2.71828);

writeln(r1);
writeln(r2);
writeln(r3);
```

the declaration of `r1` will provide space for `r1` but perform no initialization whatsoever. That is, default initialization of its fields will be inhibited. In contrast, the declaration of `r2` will result in all three fields being default-initialized. (The `noinit` flag is passed as false to each scalar type's constructor, but since no value is supplied, the default value for that type is used instead.) The declaration of `r3` will result in fields `i` and `u` remaining uninitialized while `r` is set to the real value supplied. The output of the program should be

```
(i = 0, r = 0.0, u = 0)
(i = <rj>, r = 2.71828, u = <rj>)
```

where `<rj>` means "random junk".

A Appendices

A.1 List of Proposed Changes

This appendix contains a list of the proposed changes to the Chapel specification, gleaned from the body of the proposal. This is followed by a prioritized list of tasks to be pursued in their implementation.

A.1.1 Proposed Implementation Changes

Here is a synopsis of the proposed changes, in approximate priority order.

Use Copy-Constructors – in place of assignment in initialization. This is a UMM requirement.

Rename initCopy – For consistency, mostly.

Simplify defaultOf – Rework the implementation of `_defaultOf()` so it takes precedence over the user-defined default constructor (and calls it unless overridden).

Unify Initialization – Reduce the initialization logic to a single conditional: Use the explicit initializer if present; otherwise, call `_defaultOf()`.

Separate Allocation – Pull allocation out of class constructors and get rid of the `meme` argument passed to the all-fields constructor.

Constructors as Methods – To support in-place initialization.

Add Initializer Clause – To control field initialization in the context of a constructor.

Remove Zero-Initialization – Assuming the rest of the model is implemented as specified, this is just dead code.

Collapse Copy-Constructors – Where the argument is a new-expression containing an object of the same type. This is just an optimization.

Move RTTV to Modules – Compiler creation of runtime type values is hidden and hard to understand. This would move those functions out to module code verbatim, to aid maintainability.

Normalize Type Inference – Separate type inference in initializations and function return value construction from the creation of initialization values. Type information would be carried through to resolution rather than being squashed out of the AST during normalization. Advantages: Simplify type inference and separate type inference from initialization. Disadvantages: Major implementation change.

Add ctor keyword – and optional naming, to better support generic programming.

Field Extraction – Some AST could be simplified if a field-extraction primitive were supported.

A.1.2 Proposed Changes to the Specification

Proposed changes to the specification are listed below:

- Review initialization story w.r.t. fundamental types to ensure that it is accurate under the new mode.
- Add description of the initializer list to Classes and Records chapters.
- Describe visibility of the all-fields constructor (hidden if any user-defined constructors are present.
- Revise the description of initialization of Classes and Records in terms of a call to either `_defaultOf()` or the copy-constructor.
- Describe in an implementor's note that copy-construction can be collapsed out when the initializer expression is a constructor call.
- Review the description of allocation w.r.t. class object construction, and revise if necessary. It should allow the implementation to pull allocation outside of class constructors (including the all-fields constructor).
- Review the initialization story for Classes and Records and ensure that it does not require default- or zero-initialization to be applied before field initialization.
- Deprecate the `initialize` method.

A.1.3 Open Questions

- Should we keep the `with` keyword and parentheses?
- Provide syntax for unnamed constructor introduced with a qualifier.
- Syntactic sugar for piping constructor args through to like-named fields.
- Add syntax to explicitly take field default (just the field name).

A.1.4 Implementation Priorities

This section presents the implementation tasks in tabular form.

Task	Need	Cost	Explanation
Use Copy-Constructors	High	Low	Required for UMM. Change AST generation and chase down and eliminate internal dependencies.
Rename initCopy	Med	Med	More consistent object model increases usability, chances for adoption. Cost here is in understanding the distinction between initCopy and autoCopy and coming up with a constructor model that accommodates both. Plus the usual internal dependencies. Additional benefit is removing special-case code (pragma ”init copy fn”, etc.).
Simplify defaultOf	Med	Low	Rewrite `_defaultOf()` implementation to give it priority over the user-defined zero-args constructor for consistency and simplicity in the code.
Unify Initialization	Med	Low	Mostly in place and a clean-up task suggested to Lydia. Enables the removal of zero-initialization.
Separate Allocation	Med	Low	Means getting rid of the `meme` argument. Makes all constructors uniform and enables the Constructors as Methods task.
Constructors as Methods	Med	High	More consistent object model. Simplifies initialization. Supports in-place initialization (performance). Major change in how initialization is done. May depend on normalized type inference.
Add Initializer Clause	High	Med	Supports UMM, object model, ref fields in objects. New code, so relatively easy compared to rework. Waiting until after constructors are methods will save having to revise this part. Guaranteed initialization in the new model depends on it, but can be worked around in the current impl.
Remove Zero-Initialization	Low	Low	Compiler perf. Redundant zero-init should be removed by CVP, but the extra code bloats the AST, slows the compiler.
Collapse Copy-Constructors	Low	Med	Optimization. Faster construction of some complex types e.g. arrays. Can let need drive the priority on this one.
Move RTTV to Modules	Med	Low	Maintainability. Current init path for arrays/domains is obscured by compiler generation of the RTTV function. Exposing in module code will simplify both compiler and impl. of those types, make impl of optimizations and noinit simpler.
Normalize Type Inference	High	High	Reverse very bad design decision. Complicates much of initialization and resolution. Big change, but probably smaller than Constructors as Methods. Low priority due to risks involved, but may be bumped based on need.
Field Extraction	Low	Low	Enable further optimization of the AST using in-place updates of record fields.
Add `ctor` Keyword	Low	Med	Better support for generic programming. Low current need.

A.2 Examples

This section contains specific examples, showing how the proposal would be expanded relative to the fundamental types provided by the Chapel language.

A.2.1 Record Variables

When `T` is a record type, we expect the following Chapel code

```
var s:R;
var t:R = noinit;
var u:R = new R();
var v:R = new R(8, 9, 10.0);
var w = new R();
var x = new R(9, 16, 0.5625);
```

to produce the corresponding AST:

```
1  var s:R
2  (call _defaultOf s) // Calls R._defaultOf()
3  var t:R
4  (call _construct_R t) // Calls R.R(noinit=true)
5  var u:R
6  (call _construct_R u) // Calls R.R()
7  var v:R
8  (call _construct_R v 8 9 10.0) // Calls R.R(8, 9, 10.0)
9  var w:R
10 (call _construct_R w) // Calls R.R()
11 var x:R
12 (call _construct_R x 9 16 0.5625) // Call R.R(9, 16, 0.5625)
```

The compiler-supplied definition for `_defaultOf` for this type is expected to look like:

```
proc R._defaultOf() {
  this.();
}
```

The implementation might supply a default constructor that looks like:

```
ctor R(param noinit:bool, u:uint = 7, i:int = -2, r:real = 0.0)
  with (u = new uint(noinit, u),
        i = new int(noinit, i),
        r = new real(noinit, real))
{ }
```

The nested constructor calls in the *initializer−clause* pass along the noinit flag to the constructor for that field type. Those in turn will replace the operand field with the input or defaulted formal if `noinit` is `false` and otherwise do nothing.

We note here that the execution of an initialization in the *initializer−clause* is unconditional, so there is no way to control whether a field is initialized explicitly vs. falling back to default initialization. The puzzle can be solved by using a where clause to implement the effects of the `noinit` flag. So another way to write the above is:

```
ctor R(param noinit=false, u:uint = 7, i:int = -2, r:real = 0.0)
  where noinit == false
  with (u = u, i = i, r = r)
{ }

ctor R(param noinit=false, u:uint = 7, i:int = -2, r:real = 0.0)
  where noinit == true
  with (u = noinit, i = noinit, r = noinit)
{ }
```

In this case, the `noinit` field initializer inhibits initialization of the corresponding field.

There is still the question of how to default to the field-default value when an explicit argument is not supplied to that formal in the constructor call. This is a more general problem related to determining whether the corresponding argument has been supplied. Providing a property with the formal to indicate this would support that, but to pick up the default value for each field would still require writing the full matrix possibilities for arguments supplied or not. The question of a better syntax is left open.

With the `_defaultOf` and constructors defined above inlined in the AST and scalar replacement applied, the result would be:

```
1  var s:R
2  ('.=' s u 7)
3  ('.=' s i -2)
4  ('.=' s r 0.0)

6  var t:R

8  var u:R
9  ('.=' u u 7)
10 ('.=' u i -2)
11 ('.=' u r 0.0)

13 var v:R
14 ('.=' v u 8)
15 ('.=' v i 9)
16 ('.=' v r 10.0)

18 var w:R
19 ('.=' w u 7)
20 ('.=' w i -2)
21 ('.=' w r 0.0)

23 var x:R
24 ('.=' v u 9)
25 ('.=' v i 16)
26 ('.=' v r 0.5625)
```

A.2.2 Class Variables

When initializing a variable of class type, the following forms are possible:

```
var c:C; c = new C(false);
var d:C = new C(true);
var e = new C(false);
var f:C = noinit;
```

According to the proposal, these would result in the following AST:

```
1  var c:C
2  ('move' c nil)
3  var tmp:C
4  ('move' tmp ('alloc' sizeof(C)))
5  (call _construct_C tmp) // Calls C(false)
6  ('move' c tmp)

8  var d:C
9  ('move' d ('alloc' sizeof(C)))
10 (call _construct_C d) // Calls C(true)

12 var e:C
13 ('move' e ('alloc' sizeof(C)))
14 (call _construct_C e) // Calls C(false)

16 var f:C
17 ('move' f nil)
```

Note that the primitive `'move'` in the above code is intended to mean a bit-wise copy. It corresponds to whatever primitive operation in the implmentation accomplishes this, so it is not tied to the deprecated `PRIM_MOVE` primitive. Reference to its replacement `PRIM_ASSIGN` was avoided, because its representation in AST logging is `'='`, which suggests assignment. Assignment in general means more than a bit-wise copy, since it may also entail releasing resources currently held by the LHS.

One may imagine that after `PRIM_MOVE` has been deprecated and removed from the compiler, the `PRIM_ASSIGN` primitive may be renamed as `PRIM_MOVE` and its representation in the AST log changed to `'move'`. That is the implementation to which this document is speaking.

A.2.3 Distribution Variables

In the current implementation, distributions are implemented as classes, so we expect the creation of distribution variables to follow the pattern outlined for class types above. Thus, for the Chapel code:

```
var a: Block(rank=2);
var d: Block(rank=2) = noinit;
var b = new Block({2..5, 2..5});
var c: Block(rank=2) = new Block({1..4, 1..4});
```

we expect the intermediate AST:

```
1  var a:Block(2,int(64))
2  (call _defaultOf a) // Calls Block(2,int(64))._defaultOf()

4  var d:Block(2,int(64))
5  (call _construct_Block d) // Calls Block(2,int(64))(noinit = true)

7  var b:Block(2,int(64))
8  var tmp0:domain(2,int(64),false)
9  var tmp1:range(int(64),bounded,false)
10 (call _construct_range tmp1 2 5)
11 var tmp2:range(int(64),bounded,false)
12 (call _construct_range tmp2 2 5)
13 (call _construct_domain_2_int64_t_F tmp0 tmp1 tmp2)
14 (call _construct_Block b tmp0) // Calls Block(2,int(64))(boundingBox = tmp0)

16 var c:Block(2,int(64))
17 var tmp0:domain(2,int(64),false)
```

```
18  var tmp1:range(int(64),bounded,false)
19  (call _construct_range tmp1 1 4)
20  var tmp2:range(int(64),bounded,false)
21  (call _construct_range tmp2 1 4)
22  (call _construct_domain_2_int64_t_F tmp0 tmp1 tmp2)
23  (call _construct_Block c tmp0) // Calls Block(2,int(64))(boundingBox = tmp0)
```

As for classes, as show above, the action of `_defaultOf` on a distribution class is to set `this` to `nil`.

I am not sure what actions would be taken here when `noinit` was passed as `true`. It is up to the author for that distribution to do something sensible in that case. It seems like initializing the environment arguments `dataPar...` can be done early at low cost. Other parts of the distribution need to be filled in before it is used. This can be done piecemeal, or through an initialization-finalization method. In either case, the finalization of the initialization must be done explicitly in the client code.

At present, we may lack the syntax to perform a pointer replacement of a `nil` array pointer with a full-fledged array. Practically, this can be accomplished by inventing a new pointer-copy primitive and calling that. This leaves the user view of array assignment as being an element-by-element replacement undisturbed while supporting partial initialization of objects containing arrays.

A.2.4 Array Variables

The process through which an array variable is constructed is similar to that for a domain. The runtime type info variable has a field that contains a copy of the domain information. The type of the structure is fully known after resolution: it contains only one field whose type is exactly `domain(2,int(64),false)`.

The call to `chpl_convertRuntimeTypeToValue()` produces a result of the array type, which is the element type `int(64)` attached to the domain type above. The runtime type info variable also conveys the dimensions of the domain. (Note that a fully-constructed domain object would convey the same information.)

In the current implementation, the function `chpl__buildArrayRuntimeType()` calls `dom.buildArray(eltType)`, which attaches the element type to the passed-in domain and returns a fully-constructed array. The `buildArray()` function in turn calls `dsiBuildArray` on the value extracted from the domain representation. The value field is the actual domain object, which is wrapped in the `_domain` record to support privatization. The actual work of building the array is done in `dsiBuildArray`; the call to `_newArray()` merely updates the fields in the container record.

The function `dsiBuildArray` calls down to the array constructor for the corresponding domain type. For a default rectangular array, this is `DefaultRectangularArr`. The generic arguments `rank`, `idxType` and `stridable` are copied from the domain on which `dsiBuildArray` is called. The elementType is supplied as an argument, and a pointer to the domain is passed to the array constructor as the `dom` argument.

There is no explicit constructor for the `DefaultRectangularArr` as it stands, so a wrapper for the all-fields constructor is created to set up the fields in the `DefaultRectangularArr` object. The generic fields include everything except the `dom` argument. The remaining fields in the object are value fields. Their types are mostly dependent upon the generic fields. Thus, the generic fields must all be resolved before the types of the remaining fields can be established.

The `initialize()` procedure is called implicitly as part of the all-fields constructor implementation. In this routine, if the field `noinit_data` is true, then allocation and initialization of the `_ddata` representation is skipped. This is used in `dsiReindex` and `dsiSlice` to create an alias of that representation (rather than creating it anew). Otherwise, the call to `_ddata_allocate` takes care of both allocating and initializing the array data. We note that at this point,

the allocation and initialization could be separated if, for example, the algorithm could guarantee that initialization were performed later.

At this point, the proposal is to keep the construction of arrays largely intact. In order to carry out the deprecation of the `initialize` function, however, it would be necessary to rewrite the pertinent forms of array constructors so that they could call the `initialize()` function explicitly.

Under this proposal, the explicit constructor for a `DefaultRectangularArr` would look like:

```
ctor DefaultRectangularArr(type eltType, param rank:int, type idxType,
                           param stridable:bool, dom: _domain)
  with (eltType = eltType, rank = rank, idxType = idxType,
        stridable = stridable, dom = dom) {
    initialize();
  }
```

As noted above, the four generic arguments establish the type of the object being constructed. Once the type is instantiated, the types of all of the fields in that type are determined. Current behavior is to default-initialize those fields. The same action will be taken in the constructor shown. The type of the domain argument was abbreviated as `_domain`. However, given the arguments to the constructor, this type could be expanded as `DefaultRectangularDom(rank=rank, idxType=i` just as it is in the class declaration.

Aside from the layers that are added by the DSI abstraction and the privatization mechanism, the representation of arrays are mostly class-based. As has been mentioned above, it should be possible to replace the runtime type implementation with appropriate introspection in the compiler. That done, in the actual implementation, the domain field in an array could be represented by a base class that was the common ancestor of all domain times. No further information would be required in the representation of the domain field in an array. On the other hand, the domain object actually attached has a runtime type associated with it. This supports the polymorphic behavior required to uniformly represent arrays based on disparate domain types. I believe that dynamic dispatch is already used to handle this polymorphic behavior, so there would be no performance loss associated with removing an explicit type from the domain field.

To implement a `noinit` capability that initializes everything except the values of the array elements, it would be necessary to supply a boolean to control whether the call to `init_elts` is made. That flag would have to be propagate back up the call hierarchy outlined above, and the `noinit` initializer translated into that flag by the compiler at the place where `chpl__convertRuntimeTypeToValue()` is currently called.

A.2.5 Domain Variables

I suspect (though I still don't understand it fully) that the "runtime type" mechanism — as used with domain variable declarations — provides the means to create a variable with a partial type, and have the rest of the type information filled in later in the code. It modifies the compiler's normal type resolution machinery, to back-patch the concrete type of the variable after the initially-lacking information is supplied.

This technique has the advantages that the full type of the may be resolved statically, and the representation of the type itself remains flat. It has the disadvantage that it adds complexity and special-case code to the compiler handle this specialized type inference. It also adds bulk to the AST and even the generated code, so the efficiency advantages of the flat representation become questionable.

An alternative approach would be to partition the representation of the type, so that information supplied in the initial declaration was stored in the top-level object. Information that was supplied later (e.g. in an initializer) would be stored

in a contained object that could be polymorphic. To represent the variable type information, the top-level structure need only contain a class variable. The object attached to that class variable at run time would determine the full type of the domain being represented.

Certain code in the compiler needs to inspect elements of the full static type of a domain. Here is where the special-case code must come into play. The compiler needs to know how to traverse the AST to extract element types that are added post-facto. I think that this kind of introspection follows a much more logical path — meaning that it would be easier to understand and maintain. Another large benefit of this approach is that code implementing the top-level `domain` type need not be expanded for each concrete type supported. The sizes of the AST and generated code would shrink accordingly.

Example AST given below assumes that the full concrete type of the domain has been computed by the compiler without specifying the method used. Any AST elements related to that type computation are omitted. It also assumes that the existing "flat" representation is used. However, for clarity, we assume that the actual construction of the object has been factored — specifically, the call to the convertRuntimeTypeToValue is used only to establish the type of the variable; construction of the object is performed through a separate call.

In that case, the Chapel code:

```
var D:domain(2);
var E = {1..2, 3..4};
var F:domain(2) = {6..7, 8..9};
var G:domain(2) = noinit;
```

will give rise to the following AST:

```
1  var D:domain(2,int(64),false)
2  (call _defaultOf D) // Calls domain(2,int(64),false)._defaultOf()

4  var E:domain(2,int(64),false)
5  var tmp1:range(int(64),bounded,false)
6  var tmp2:range(int(64),bounded,false)
7  (call _construct__range_int64_t_bounded_F tmp1 1 2)
8  (call _construct__range_int64_t_bounded_F tmp2 3 4)
9  (call _construct__domain_2_int64_t_F F tmp1 tmp2) // Calls domain(2,int(64),false)(tmp0)

11 var F:domain(2,int(64),false)
12 var tmp1:range(int(64),bounded,false)
13 var tmp2:range(int(64),bounded,false)
14 (call _construct__range_int64_t_bounded_F tmp1 6 7)
15 (call _construct__range_int64_t_bounded_F tmp2 8 9)
16 (call _construct__domain_2_int64_t_F F tmp1 tmp2) // Calls domain(2,int(64),false)(tmp0)

18 var G:domain(2,int(64),false)
19 (call _construct__domain_2_int64_t_F G) // Calls domain(2,int(64),false)(noinit=true)
```

For the domain labelled `D`, the type constructor supplies default values for the `idxType` and `stridable` type parameters. These are `int(64)` and `false`, respectively. It then calls the default (i.e. zero-args) constructor through `_defaultOf()`.

There is no difference between the initialization code for variables `E` and `F`. The internal difference is that the compiler infers the full type of `E` from its initializer, whereas for `F` the rank is provided explicitly: the compiler only has to infer the `idxType` and `stridable` type parameters. The `idxType` it infers from the initializer; for `stridable`, it uses the default value (`false`).

Initialization for `G` passes the `noinit` flag to a constructor that can receive it (most likely the "zero-args" constructor).

The compiler-generated implementation of `_defaultOf()` for domain types is expected to look like:

```
proc domain(2,int(64),false)._defaultOf() {
  this.(); // Call the zero-args constructor.
}
```

The default constructor could be written explicitly as:

```
ctor domain(2,int(64),false)(param noinit=false) {
}
```

Since the *initializer−clause* is empty, this basically means "use the default value for every field". For this domain type, it ends up setting the two ranges to empty ranges and setting the `dist` field to `nil`. Note that the `noinit` flag is ignored. This means that the domain is initialized in the same way whether or not the `noinit` initializer is supplied. Also, it means that the `pragma "noinit"` need not be specified on concrete domain implementations.

At least one of the constructors for a rectangular domain would expand to:

```
ctor domain(2,int(64),false)(r1:range, r2:range) {
  with (ranges = (r1,r2))
}
```

which means that the supplied range arguments should be used to replace the ranges in the domain structure. Since it is not mentioned in the *initializer−clause*, the `dist` field is set to `nil`.

When these definitions are inlined in the above code and scalar replacement applied, the resulting AST is expected to look like:

```
1  var D:domain(2,int(64),false)
2  var tmp0:range(int(64),bounded,false)
3  (call _defaultOf tmp0)
4  ('.=' D ranges.x0 tmp0)
5  var tmp1:range(int(64),bounded,false)
6  (call _defaultOf tmp1)
7  ('.=' D ranges.x1 tmp1)
8  ('.=' D dist nil)

10 var E:domain(2,int(64),false)
11 var tmp1:range(int(64),bounded,false)
12 var tmp2:range(int(64),bounded,false)
13 (call _construct__range_int64_t_bounded_F tmp1 1 2)
14 (call _construct__range_int64_t_bounded_F tmp2 3 4)
15 ('.=' E ranges.x0 tmp1)
16 var tmp1:range(int(64),bounded,false)
17 ('.=' E ranges.x1 tmp2)
18 ('.=' E dist nil)

20 var F:domain(2,int(64),false)
21 var tmp1:range(int(64),bounded,false)
22 var tmp2:range(int(64),bounded,false)
23 (call _construct__range_int64_t_bounded_F tmp1 5 6)
24 (call _construct__range_int64_t_bounded_F tmp2 7 8)
25 ('.=' F ranges.x0 tmp1)
26 var tmp1:range(int(64),bounded,false)
27 ('.=' F ranges.x1 tmp2)
28 ('.=' F dist nil)

30 var G:domain(2,int(64),false)
31 var tmp0:range(int(64),bounded,false)
32 (call _defaultOf tmp0)
33 ('.=' G ranges.x0 tmp0)
34 var tmp1:range(int(64),bounded,false)
```

```
35   (call _defaultOf tmp1)
36   ('.=' G ranges.x1 tmp1)
37   ('.=' G dist nil)
```

An interesting observation here is that the AST still forces us to create an unnamed range literal and then copy it into the appropriate field in the domain. If the AST instead supported the creation of a field reference, then fields in a structure could be updated in-place. This could be used in some places to avoid bitwise copies (such as that performed by the `PRIM_SET_FIELD` primitive. In that case, for example, initialization of the first range on lines 2-4 above could be replaced by:

```
1  var tmp0:ref 2*range(int(64),bounded,false)
2  ('move' tmp0 ('.' D ranges)) // Here '.' means extract field reference.
3  var tmp1:ref range(int(64),bounded,false)
4  ('move' tmp1 ('.' tmp0 x0))
5  (call _defaultOf tmp1)
```

A.2.6 Sync and Single Variables

For the Chapel code:

```
var a: sync int;
var b: sync int = noinit;
var c: sync int = n;

var d: single int;
var e: single int = noinit;
var f: single int = n;
```

the proposed implementation would give the following AST:

```
1   var a:_syncvar(int(64))
2   (call _defaultOf a) // Calls _syncvar(int(64))._defaultOf().

4   var b:_syncvar(int(64))
5   (call _construct__syncvar_int64_t b) // Calls _syncvar(int(64))(noinit=true).

7   var c:_syncvar(int(64))
8   (call _construct__syncvar_int64_t c n) // Calls _syncvar(int(64))(i:int).

10  var d:_singlevar(int(64))
11  (call _defaultOf a) // Calls _singlevar(int(64))._defaultOf().

13  var e:_singlevar(int(64))
14  (call _construct__singlevar_int64_t b) // Calls _singlevar(int(64))(noinit=true).

16  var f:_singlevar(int(64))
17  (call _construct__singlevar_int64_t c n) // Calls _singlevar(int(64))(i:int).
```

This rendition assumes that the `_syncvar` and `_singlevar` types have been rewritten as records to take advantage of automatic allocation and deallocation within the compiler. Blank intent for those types would have to be adjusted to `ref` to preserve the existing semantics. A concrete `ref` intent makes no attempt to read the value stored in the sync var (e.g. using `readFE`). The AST for the current class-based implementation would include explicit allocation for each of the variables and the assumption that they are properly deallocated at the end of the scope in which they are declared. Mention of a class variable in the AST includes an implicit dereferencing, consistent with current behavior.

The compiler-generated `_defaultOf()` for the sync type would look like:

```
proc _syncvar(int(64))._defaultOf() {
  this.(); // Call the zero-args constructor
}
```

which serves just fine. The code for the constructor that can accept zero arguments might look like:

```
ctor _syncvar(?t).(param noinit=false, val=t._defaultOf()) {
  with (value=val)
  __primitive("sync_init", this);
}
```

The current implementation uses the special pragma "ignore noinit", which is no longer needed because the noinit flag is simply ignored.

When constructors are implemented to the point of supporting *initializer−clause*s, the pragma "omit from constructor" will also be unnecessary because the all-fields constructor would not always be called from an explicit constructor as it is today. Instead, field-initialization would be covered by the semantics of the *initializer−clause*. The all-fields constructor would be called only if invoked explicitly from user code (impossible if any user-defined constructor is present), or to implement the semantics of `_defaultOf()`.

The implementation for single variable would, of course, parallel the implementation for sync vars. When the `_defaultOf()` and the constructor calls are inlined, the above as would simplify to:

```
1  var a:_syncvar(int(64))
2  ('.=' a val 0)
3  ('sync_init' a)

5  var b:_syncvar(int(64))
6  ('.=' b val 0)
7  ('sync_init' b)

9  var c:_syncvar(int(64))
10 ('.=' c val n)
11 ('sync_init' c)

13 var d:_singlevar(int(64))
14 ('.=' d val 0)
15 ('single_init' d)

17 var e:_singlevar(int(64))
18 ('.=' e val 0)
19 ('single_init' e)

21 var f:_singlevar(int(64))
22 ('.=' f val n)
23 ('single_init' f)
```

A.2.7 Tuple Variables

For the Chapel code:

```
var a:2*real;
var b:2*real = noinit;
var d:2*real = (12.0,13.3);
var e = d;
```

the proposed implementation would give rise the the following AST:

```
1  var a:2*real(64)
2  (call _defaultOf a) // Calls 2*real(64)._defaultOf().

4  var b:2*real(64)
5  (call _construct_2_real64_t b) // Calls 2*real(64).2*real(64)(noinit=true).

7  var d:2*real(64)
8  (call _construct_2_real64_t d 12.0 13.3)

10 var e:2*real(64)
11 (call _construct_2_real64_t e d) // Calls the copy-constructor.
```

The compiler-generated `_defaultOf()` for a 2-tuple of reals would look like:

```
proc 2*real(64)._defaultOf() {
  this.(); // Call the zero-args constructor (see below).
}
```

This raises a bit of a design issue: The compiler-generated version of `_defaultOf()` wants to call a constructor as a method on `this`, but we don't really have syntax for that. I normally expect **new**˜*type-name* on a record type to produce an anonymous object, so adding a `'new'` keyword is not the right choice. I chose to omit the type name, but supplying it explicitly is equally valid.

The problem is that calling a constructor on an object that has already been initialized (as we expect all objects to be) is normally illegal. Even going so far as to supply syntax for calling a constructor on an existing object seems like bending the rules. On the other hand, we know (special case) that the operand of a `_defaultOf()` method is uninitialized, so at least in that context the constructor call is semantically correct.

The syntax problem is created by the existence of `_defaultOf()`, which is really just a synonym for a call to the zero-args constructor. After more of the constructor story is in place, we can consider deprecating `_defaultOf()` as an approach to avoiding the syntax problem. In the mean time, it is acceptable to treat the body of `_defaultOf()` as a special case, and allow constructor calls (only on `this`) in that syntactical context only.

The code for the compiler-generated zero-args constructor (supporting the noinit initializer) would look like:

```
ctor 2*real(64)(param noinit = false) {
  if noinit then { }
  else { this.x0 = 0.0; this.x1 = 0.0; }
}
```

The code for the compiler-generated all-fields constructor (called on line 8) would look like:

```
ctor 2*real(64)(a0:real(64), a1:real(64)) {
  this.x0 = a0; this.x1 = a1;
}
```

After inlining and scalar substitution, the AST would reduce to:

```
1  var a:2*real(64)
2  ('.=' a x0 0.0)
3  ('.=' a x1 0.0)

5  var b:2*real(64)

7  var d:2*real(64)
8  ('.=' a x0 12.0)
9  ('.=' a x1 13.3)

11 var e:2*real(64)
12 ('.=' a x0 ('.' d x0))
13 ('.=' a x1 ('.' d x1))
```

A.2.8 Scalar Variables

For scalar types, we would expect declarations of the form:

```
// type T = int(64);
   var a:T;
   var b:T = noinit;
   var c:T = new T("42");
   var d = 13; // Initializer expression e has type T.
   var e = d; // Same.
```

to produce the following AST (after resolution):

```
1   var a:int(64)
2   (call _defaultOf a) // Calls int(64)._defaultOf().

4   var b:int(64)
5   (call _construct_int64_t b) // Calls b._construct_int64_t(noinit=true).

7   var c:int(64)
8   var tmp:int(64)
9   (call _construct_int64_t tmp "42") // Calls tmp._construct_int64_t(s:c_string).
10  (call _construct_int64_t c tmp) // Calls c._construct_int64_t(i:int(64)).
11  // === OR === //
12  (call _construct_int64_t c "42") // Calls c._construct_int64_t(s:c_string) in situ.

14  var d:int(64)
15  (call _construct_int64_t d 13) // Calls the copy-constructor.

17  var e:int(64)
18  (call _construct_int64_t e d) // Calls the copy-constructor.
```

This looks a lot more complex than one would expect, but that is for consistency with more complex types. Scalar types are really a special case because they exhibit value semantics and have trivial constructors, destructors and assignment operators. When that is true, copy-construction is equivalent to assignment is equivalent to a move operation.

The definition of the various constructors provides the semantics you would expect. On line 2, the `_defaultOf()` method on a 64-bit integer replaces the LHS with a zero of that size. If `=noinit` is a valid initializer for an object of type `int(64)`, then at least one of its constructors will accept a `param bool` to control its actions. For example, the default- and copy-constructor for int(64) could be covered by:

```
ctor int(64)(val = 0, param noinit = false) {
  if noinit then { }
  else { this = val; }
}
```

The constructor does not contain a field-initializer list, because the int(64) type has no sub-fields.

The promotion constructor from type `c_string` to type `int(64)` does not currently exist, but would presumably do something logical, like converting a text representation of an integer into the internal representation of an integer and then overwriting the receiver with that value, for example:

```
ctor int(64)(s:c_string) {
  this = __primitive("convert_c_string_to_int64", s);
}
```

The first form (lines 9-10) would be without the optimization that collapses out unnecessary copy-constructors: the second form (line 12) would be with that optimization applied. The last two cases are simple copy-constructor calls.

After the compiler has inlined the calls to `_defaultOf()` and the various constructors, and replaced scalar assignment with move (or assign) primitives, the above AST would devolve to:

```
1     var a:int(64)
2     ('move' a 0)

4     var b:int(64)

6     var c:int(64)
7     ('move' c ('cast' 0x42)) // Cast from a char literal to int(64).

9     var d:int(64)
10    ('move' d 13)

12    var e:int(64)
13    ('move' e d)
```

A.3 Deferred Array Initialization

In the body this design document, the proposed support for a `noinit` initializer leaves it up to the type designer to define what "noinit" initialization means and the steps that must be taken to move an object from the "noinit" state into a fully-initialized state.

However, since the `noinit` feature was proposed specifically with array initialization in mind, Additional detail should be provided to show that the general feature can be used to implement an array type that exhibits the specific behavior that was originally envisioned.

The noinit feature was intended to reduce the cost of creating an array, by avoiding element-by-element initialization in cases where the algorithm immediately fills the array with valid values from another source. In that case, the default-initialization of the elements is wasted effort, and only the algorithm designer really knows this. It is possible that an optimizer could identify and remove useless default-initializations. But providing explicit means to disable this default-initialization is important for three reasons:

1. That optimization has not yet been implemented;
2. Even if it were, some cases might slip through; and
3. Most likely, the optimization would not be able to track default-initializations that crossed function calls.

A second requirement that has emerged more recently is to ensure that when initialization is complete, it will be done in a manner that takes advantage of localized caching in the memory hierarchy. Whereas addresses in main memory may be treated uniformly within a computer node, the cache assignment of a block of memory may be determined by when it is first written. Therefore, ideally, the initialization of a segment of a distributed array should be done through the same processors as will be used in accessing that data later.[1]

Basically, we wish to defer initialization of the elements of an array until the traversal is made that writes their true initial values. Depending on whether mere allocation is considered an access by the caching system, we may also desire to defer allocation of an array segment until some attempt is made to initialize it. This strategy should fulfill both requirements.

A.3.1 Observations

An element of the design for the general "noinit" feature is that the definition of the noinit-initialized state of an object is left to the designer of that type. To stay within the bounds of this design, the only control an array implementation has over element-by-element initialization (at the point in time that the array is populated) is whether the `noinit` flag is passed to the constructor for each element.

No-initialization and default-initialization are not the only options. If

```
var x = new T(...);
```

[1] This assumes that the distribution of the array conforms to some notion of locality expressed by or used by the algorithm being run.

is the general Chapel idiom for creating an object `x` of type `T` and initializing it by calling a constructor of `T` passing in the arguments of the apparent constructor call. The = here does not represent assignment. Therefore, if the type of an array can be represented abstractly, we can invoke an array constructor and pass in arbitrary arguments. That would effectively enable complete generality in how arrays are initialized. Selecting useful ones from among these and elevating to the language level through the addition of syntactical sugar lies outside the scope of this discussion.

Default-initialization of the elements of an array at declaration time requires that the entire array be allocated and then the element-by-element initialization performed. As hinted at above, "noinit" initialization can avoid not only the element-by-element initialization but also allocation required to support it.

However, deferred initialization can be performed even if `noinit` is not specified, and the expected semantics of the language preserved. It is only necessary to store in the representation of the array, perhaps at several levels, sufficient state information to determine whether a local section or an element thereof has been initialized. The implementation of the array as an ADT can hide the details of the allocation and initialization. Up to the point that a value is explicitly stored in a give location, the implementation can respond "as if" the allocation and initialization had been done, but without performing either of those actions in fact.

A.3.2 An Example Design

Given those observations, it seems reasonable to implement deferred allocation-and-initialization at the granularity of the local subarray, whether or not the `noinit` keyword is supplied as the initializer-expression for the array as a whole. The implementation only needs to arrange that the `noinit` specifier be propagated to the subarrays at the point in time that they are allocated, to control the element-by-element initialization.

For the example design to be complete, we have to describe what we expect the behavior to be when reading from and writing to an element in an array, for the two cases where the `noinit` initializer is specified and not. The behavior includes both the functional semantics – being effects observable within the program itself – and underlying behaviors that mostly affect performance.

Most of this can be demonstrated through the implementation of element access functions (read and write). But to demonstrate that the NUMA performance requirements are met, we also have to show some detail of how iteration may proceed over both noinit-initialized and default-initialized arrays.

A.3.3 Example Code

This section give example code for the implementation of an array type supporting the `noinit` initializer. Commentary is inserted between the code sections, to explain their expected behavior and how they conform to the design requirements.

> *Implementors' note*. This example uses the latest syntax for implementing the semantics of `noinit` initialization using a specialized constructor as presented at the deep-dive. This new syntax has not yet been back-patched into the main document.

A number of simplifications are made in comparison with the existing implmentation:

- Array aliases and reference counting is not supported.

- Privatization is not supported.
- Bulk copying of array elements is not supported.
- Remote value forwarding is not supported.
- Shifted indexing is not supported.

A.3.3.1 Array Default Construction

Given:

```
confi const n = 10;
var D = {1..n};
```

the three declarations

```
var A: [D] real;
var B: [D] real = noinit;
var C: [D] real = 3.1416;
```

will be linked by the compiler to three different array constructor calls. The first will follow the existing path for array construction by calling the zero-args constructor; the second will call the noinit constructor; the third will call an overloaded version of the constructor that accepts a constant initializer value.

In the current implementation, the array constructor called by the compiler through the call chain `chpl__buildArrayRuntimeType()` –¿ `_domain.buildArray(eltType)` –¿ `dom.dsiBuildArray(eltType)`. The implementation for `DefaultRectangularDom.ds` currently looks like:

```
proc DefaultRectangularDom.dsiBuildArray(type eltType) {
  return new DefaultRectangularArr(eltType=eltType, rank=rank, idxType=idxType,
                                   stridable=stridable, dom=this);
}
```

To support the `noinit` feature, it is probably simplest to add a flag that is passed down through the call chain to the point where the array constructor is actually called. To also support initialization to a particular value, we can consider passing along an initializer value. At some point in the call chain, a default value can be computed from the `eltType`; from there on down, the default value would be passed along verbatim. To incorporate the two new ways of initializing arrays, the above constructor call would be modified to read as follows:

```
proc DefaultRectangularDom.dsiBuildArray(type eltType, isNoinit, initVal) {
  if isNoinit then
    return new DefaultRectangularArr.ntor(eltType=eltType, rank=rank, idxType=idxType,
                                          stridable=stridable, dom=this, initVal=initVal);
  else
    return new DefaultRectangularArr(eltType=eltType, rank=rank, idxType=idxType,
                                     stridable=stridable, dom=this);
}
```

It is important to note that we need some syntax for calling the noinit constructor under program control. The compiler must therfore recognize the noinit constructor `ntor` as a method of the class or record and provide a name that can be used to invoke it.

Moving back up the call chain, we would have:

```
  proc _domain.buildArray(type eltType, isNoinit, initVal) {
    return _value.dsiBuildArray(eltType, isNoinit, initVal);
  }

pragma "runtime type init fn"
proc chpl__buildArrayRuntimeType(dom: domain, type eltType,
                                 isNoinit = false, initVal = _defaultOf(eltType))
  return dom.buildArray(eltType, isNoinit, initVal);
```

Since `chpl__buildArrayRuntimeType()` is called by the compiler, we put the default values for `isNoinit` and `initVal` in its definition, so we can leverage the compiler's support for default arguments. Other than that, there is no magic involved: we simply pass initialization parameters down the call chain to the place where they can be used.

Now let us examine how a normally-initialized DefaultRectangularArray would be constructed. Currently, only the compiler-supplied all-fields constructor is used. Therefore, we need to look at the declaration for the class to compose an explicit constructor that will do the same thing.

```
class DefaultRectangularArr : BaseArr {
  type eltType;
  param rank : int;
  type idxType;
  param stridable: bool;

  var dom : DefaultRectangularDom(rank=rank, idxType=idxType,
                                  stridable=stridable);
  var off: rank*idxType;
  var blk: rank*idxType;
  var str: rank*chpl__signedType(idxType);
  var origin: idxType;
  var factoredOffs: idxType;
  var data : _ddata(eltType);
  var noinit_data: bool = false;
```

We must also take into consideration the `initialize()` function. It is called implicitly by the compiler currently. Support going forward can be obtaind by calling it explicitly instead. We will pull out the noinit_data test, as this is precisely the choice we want to make for default- versus noinit-construction.

```
  proc initialize() {
    for param dim in 1..rank {
      off(dim) = dom.dsiDim(dim).alignedLow;
      str(dim) = dom.dsiDim(dim).stride;
    }
    blk(rank) = 1:idxType;
    for param dim in 1..(rank-1) by -1 do
      blk(dim) = blk(dim+1) * dom.dsiDim(dim+1).length;
    computeFactoredOffs();
    var size = blk(1) * dom.dsiDim(1).length;
    data = _ddata_allocate(eltType, size, initVal);
  }
```

When initialization takes place, the passed-in initial value is passed along to `_ddata_allocate`, so it can use that to initialize elements. The definition of `_ddata_allocate` is modified accordingly.

```
inline proc _ddata_allocate(type eltType, size: integral) {
  var ret:_ddata(eltType);
  __primitive("array_alloc", ret, eltType, size);
  init_elts(ret, size, eltType);
  return ret;
}
```

```
proc init_elts(x, s, type t, initVal) {
  for i in 1..s {
    pragma "no auto destroy" var y: t = initVal;
    __primitive("array_set_first", x, i-1, y);
  }
}
```

After the preliminaries, we are able to write:

```
ctor DefaultRectangularArr(type eltType, param rank:int,
                           type idxType, stridable:bool,
                           dom, initVal)
with (eltType=eltType, rank=rank, idxType=idxType,
      stridable=stridable, dom=dom, initVal=initVal)
{
  initialize();
}
```

To the class, we will have to add one field, to store `initVal`. The field `noinit_data` can be removed, as it is no longer needed. Being class variables, the ddata fiels used to store the array valueswill not be allocated until initialize() is called. The types of all of the fields following `dom` up to and including `data` are established when the generic DefaultRectangularArr type is instantiated during resolution. They are not mentioned in the initializer-list, since we can accept their default-initialized values. There is no particular reason to force any of them to remain uninitialized, unless it happened that the chosen `idxType` were expensive to initialize.

We may as well proceed to the description of the noinit constructor for a `DefaultRectangularArr`, and then consider how this design might be back-patched into the default initializer to defer allocation and initialization of the contained elements.

A.3.3.2 Array Noinit Construction

To implement noinit initialization for the default rectangular array, all we need to do is

```
ntor DefaultRectangularArr(type eltType, param rank:int,
                           type idxType, stridable:bool,
                           dom, initVal)
with (eltType=eltType, rank=rank, idxType=idxType,
      stridable=stridable, dom=dom, initVal=initVal)
{}
```

This is identical with the constructor except that the call to initialize() has been omitted. The interesting part is how an element of the array is initialized later in the code. From the view of the user, the array has been initialized, but a read access of any element in the array yields its default value.

Array value accesses are implemented through the `this` routine in ChapelArray.chpl, which returns a reference to the indexed element. To perform the magic of faking in the default value on a read before the corresponding element has been initialized, we need to be able to distinguish a read from a write. We add a param bool `setter` to the signature of `this()`, and assume that the compiler sets this appropriately, depending on whether the array access is evaluated as an lvalue or rvalue.

At the array level, `this()` is modified to pass its value on to `dsiAccess()`. We could also consider splitting `dsiAccess` into `dsiGet()` and `dsiSet()`.

```
inline proc this(i: rank*_value.dom.idxType) ref {
  // Simplified.
  return _value.dsiAccess(setter, i);
}
```

In the array implementation, we can check this flag to do something different for reads vs. writes. In the case of a read, the behavior shown below is that the initialization value stored in the array implementation is returned for an access made before the contents of the array are initialized.

When an attempt is made to write into the array, the entire array is allocated an initialized in bulk using the supplied initialization value.

```
inline proc dsiAccess(param setter: bool, ind : rank*idxType)
where setter == false
{
  // If the data array is uninitialized, just return the default value.
  if data == nil then
    return initVal;
  else {
    var dataInd = getDataIndex(ind);
    return data(dataInd);
  }
}

inline proc dsiAccess(param setter: bool, ind : rank*idxType) ref
where setter == true
{
  // Allocate and initialize the array the first time an attempt is made to write into it.
  if data == nil then
    initialize();
  var dataInd = getDataIndex(ind);
  return data(dataInd);
}
```

This design still suffers from the disadvantage that the entire array is allocated and every element initialized in bulk. Supposing that we are expecting to overwrite the array after it is initialized, we can have a different form of `initialize()` that only allocates and does not initialize. We can factor the element initialization out of `_ddata_allocate()`:

```
inline proc _ddata_allocate(type eltType, size: integral) {
  var ret:_ddata(eltType);
  __primitive("array_alloc", ret, eltType, size);
  return ret;
}
```

Then, `initialize()` can be modified to leave out initializing the array elements.

```
proc initialize(param initialize_elts = true) {
  for param dim in 1..rank {
    off(dim) = dom.dsiDim(dim).alignedLow;
    str(dim) = dom.dsiDim(dim).stride;
  }
  blk(rank) = 1:idxType;
  for param dim in 1..(rank-1) by -1 do
    blk(dim) = blk(dim+1) * dom.dsiDim(dim+1).length;
  computeFactoredOffs();
  var size = blk(1) * dom.dsiDim(1).length;
  data = _ddata_allocate(eltType, size, initVal);
  if (initialize_elts) then
    init_elts(data, size, eltType, initVal);
}
```

Then our setter version of this above can be modified to call `initialize(false)`. This will allocate the array and fill it with potentially garbage values. That is probably not what the user desires, but this is proof that we can do it.

Going back to the original idea of noinit, where delayed initialization is to be applied to the elements rather than to the array as a whole, we can perform the just the allocation part by calling `initialize(false)` in the body of the `ntor` and calling `initialize(true)` in the body of the ctor.

A.3.3.3 Parallel Initialization

A.3.3.4 Class Elements

A.4 Usage Notes

This appendix contains notes on usage, related to certain design choices.

A.4.1 Running Code Prior to Construction

In some cases, when creating a new object it is desirable to run code prior to construction. No specific provision has been made in the specification for constructors to support this. In particular, the initializer list supports the initialization of each field individually (in an arbitrary order), but does not support initializing many of them as the result of one function. How can this case be handled?

In Chapel, this is especially easy, given its support for generic functions and overloading. The solution is to wrap object creation within a factory method. The one restriction that must currently be obeyed is that the result of such a factory method must be described by a single type. If the result type is not generic (i.e. it is a single static type) it may still be polymorphic (i.e. the returned object may have one of several runtime types).

For example:

```
class B { }
class D1 : B { }
class D2 : B { }

proc D_factory(i:int) : B {
  select i {
    when 1 do return new D1();
    when 2 do return new D2();
    otherwise compilerError("Bad D_factory argument.");
  }
}
```

This example is as simple as possible. But can easily be imagined that `D_factory` accepts multiple arguments and performs nontrivial computations, passing the appropriate arguments to the constructor corresponding to each case. Common code can be placed above the select statement, and case-specific code within the statement controlled by each `when`.

The only potential down-side of this approach as compared to providing more general semantics within the constructor itself is that — at least on the surface — the initializers for some fields must be built up outside the constructor and then passed in as arguments. That in turn may appear to necessitate a copy.

Many of these apparent drawbacks may be dismissed due to the fact that the same copy would be required even if the initializer value were generated within the constructor itself. The remaining cases may be answered with the eventual inclusion in the compiler implementation of copy-avoidance. Specifically, if the coder avoids giving a name to the initializer expression passed to a constructor, the compiler may complete the initialization of the corresponding field with nothing more expensive than a pointer-copy operation.

The result type may also be generic. That observation gives the user considerably more flexibility. It means that the types of the results of specific cases within the factory method must have the same name. Otherwise, they need have little in common. A wrapper type containing a type field can handle the most general case.

```
record Q { }
record R { }
record S { type T; }

proc S_factory(i:int) : S {
  select i {
    when 1 do return new S(T=Q);
    when 2 do return new S(T=R);
    otherwise compilerError("Bad S_factory argument.");
  }
}
```

Here again, the example is as simple as possible. However, it is easy to imagine considerable computation preceding the call to new, and those arguments being fed to the constructor for each type being constructed.

A.4.2 Escape of Partially-Constructed Objects

There was some discussion around having a second body in the definition of the constructor — the first body taking the object from a field-initialized state to a fully-constructed state, the second containing actions that are permissible on the fully-constructed object (e.g. calling methods within the same type).

I believe the discussion lapsed without coming to a conclusion. However, it may now be argued that the person writing a constructor should know the point at which an object of that type is fully-initialized. At the same time, without external knowledge the compiler has no way to determine whether the division between the two bodies is correct. Therefore, there is no need set this point off specially in the syntax.

Carrying this argument further, the author of a constructor should know fully the interface and effects of any function he calls during construction (more so than the application programmer). The problem of preventing the early escape of a partially-constructed object is pushed back on the author. The class author himself must be the best judge of when it is safe to publish an object of that type.

One can see that the same problem arises whether a constructor has one body or two. In the latter case, the author must be aware of the danger that calling a certain routine poses if he chooses to place a call to it in the second body. That establishes the premise that he knows well enough how to make the calls in the right order so as to prevent early escape.

The availability of a second body may help the author consider the order in which things must be done, but it cannot force him to do this correctly. For that matter, there is nothing special about the number 2 in this context. We may as well support any number of bodies, including zero.