ODL: Preliminary Language Report

Doug Lea
SUNY at Oswego / NY CASE Center
`dl@g.oswego.edu`

Draft 4: 17 September 1993

1 Overview

ODL (*Our Design Language*) was devised for use in presenting object-oriented system design concepts in a programming language- and system-independent manner. However, the language may be useful in its own right. ODL is in most respects a typical object-oriented programming language, containing constructs corresponding to the notions of classes, attributes, methods, objects, and links, but geared towards distributed programs. Programs are written by defining classes, operations, and supporting type and execution information.

2 Names and Scopes

ODL declarations consist of named classes, parameterized classes, records, fields, slots, and constraints. The constructs `class`, `op`, `fn` and `record` introduce named scopes.

Class declarations may be nested within others. Class names introduced at the same scope level must be unique. Name resolution for embedded classes, fields and slots follows most-closely-nested rules but may be redirected via the *name*`::` operator, where *name* is a named scope introducer. Any name may be qualified as `local`, which disables reference outside of the current scope except within constructor expressions.

The top-level declarations of an entire ODL program are considered to be encased within `class System` ...`end`. Class `System` must contain single user-defined operation `main`. An ODL implementation must support a command equivalent to `(new System(`...`)).main(`...`)` that constructs object `system` and initiates execution.

Normally, names must be declared before they are otherwise referenced. However, multiple declarations may be listed in mutually referential fashion by enclosing them within `letrec` ...`end`. Also `generator` statements may mention as-yet undeclared classes.

3 Types

There are four kinds of value types: scalars, links, arrays, and records. Values of these types are not themselves objects. A fifth kind, that of code bodies defining slots is a "second-class" type, discussed separately below.

The scalar types are `bool`, `int`, `real`, `char`, `time`, and `blob`. Literals for each type are self-describing. The first four have the usual meanings. Restrictions on the bounds and precision of implementations of these types are permitted but not otherwise defined. The `time` type denotes times. It is left unspecified whether `time` is a discrete or continuous quantity. The type `blob` describes completely opaque values that may have implementation-dependent meaning.

A link is a value denoting the identity of an instance. Link types are introduced via the declaration of classes of the same name. Their subtype structure mirrors the declared inheritance relations among classes. Other scalar types do not obey any subtype structure. They are distinct, incommensurate types in ODL.

Fixed-bound vector values are designated via postfix `[`*capacity*`]`. Elements are referenced using subscripts.

Records are named tuples of values. Records define fields of any of the four kinds of value type. Nested fields are referenced via dot notation. Any field declaration may be qualified as:

`common` The value must be the same across all instances of this record.

`unique` The value must differ across all instances of this record.

`opt` The value need not be present. A record declaration with an `opt` field is considered equivalent to two declarations, one with the field, and one without it. A declaration with two `opt` fields is equivalent to four, and so on. However, syntactically these versions may be described and used as an aggregate. Absence of a value may be indicated via `null` and discerned via `null(`*field*`)`. Access to a nonpresent `opt` value may be trapped as a static program error, dynamic execution error, or left undetected by translators.

All combinations of these qualifiers are semantically meaningful and allowed; for example, even the odd combination of `common` and `unique`, which is an awkward way of saying that only one value is possible.

Record values are expressed by naming the record type followed by a parenthesized list of values for each field. Both positional and named syntax is allowed, but all positional values must precede named ones, fields indicated positionally may not be listed also as named, and names may not be listed more than once. Lack of value for an `opt` field need not be listed at all in the named syntax. For example, all three expressions denote the same value in:

```
  class A ... end;
```

```
record r(a: int, b: char, c: real, d: opt A);

r(1, '2', 3.0, null)
r(b := '2', a := 1, c := 3.0, d := null)
r(1, '2', c := 3.0)
```

4 Classes

Classes describe families of objects, all of which possess the same structural features. Each object is viewed as a map from a unique identity to a set of names. The map is the same for all direct instances of a given class. These names in turn map to *slots*, which may be different for each object. Unlike record fields, slots are computationally defined. There are three kinds of slots:

- One-way operations, expressed as `op m(args)`. One-way operations are those that accept a message and perform some computation independently of the message sender.
- Procedural operations, expressed as `op p(args) : replyName(args)` or `op p(args) resultName : resultType`. The client of a procedural operation waits for the recipient to reply before proceeding.
- Functional operations, expressed as `fn f(args) : fnType`. Functional operations are special side-effect-free forms of procedures. Functions may be defined concretely either via code bodies or via a special form indicating that a single value is "stored".

All slot declarations within a class are processed as if encased within `letrec` ...`end`. Slot names and arguments normally correspond to message names and fields that are accepted by instances of the class. However, internal `local` versions of any of the above may also be declared. Groups of `local` slots may be declared within `locals` ...`end`.

4.1 Concrete Definitions

Concrete definitions may be bound to slots via {...}. A special form, `<>` is used instead to indicate that a slot is bound to a concrete definition only upon construction. It is used only for "stored" functional slots that access a single value that must be intialized upon construction.

Concrete slot definitions may contain the following constructs:

Message Sends: One-way sends of messages to specified recipients.

Procedure Invocations: Blocking call/return style interaction.

Function Invocations: Side-effect-free procedural interaction.

Local Invocations: Internal sequential processing.

Rebindings: Assignment statements rebind non-`fixed` stored slots to different values via `f :=` *exp*.

Locals: `local` declarations of new transient slots maintained only within an operation.

Control flow: `if` and `while` statements controlled by boolean value expressions.

4.1.1 Message Sends

Computation is based on message passing. Objects are autonomous single-threaded computational agents that send messages listed within operations corresponding to messages from other objects. A *message* is a record that corresponds to a declared operation. ODL `op` declarations define records with names identical to operation names, and fields corresponding to argument lists. There are five phases in any act of message passing:

Invocation. An invocation listed in the concrete definition of a sender is issued.

Reception. ODL does not specify the mechanisms controlling how messages are issued and received by objects except in the assumption that they do not interfere with explicitly defined processing. Synchronicity between sending and receiving a message is neither required nor precluded.

Binding. A message is linked with a corresponding operation or version of an operation. Linkage may be determined either statically or dynamically. (Run-time binding is necessary when there are argument-based guard conditions.) Binding failures cannot occur in correct programs.

Acceptance. An accepted message is "consumed", and causes the triggering of an operation when its guard has cleared.

Execution. Computation proceeds by noninterruptibly processing all actions defined in the corresponding concrete operation.

The simplest form of a concrete operation is a sequence of one-way message sends, for example, { `a.m1(x); b.m2(y, z);` ...}. Invocations are defined by by naming them and providing field values along with a recipient designation. Instead of recipient prefixing, ODL messages may be invoked with an indication of a *class* of receivers, via *className*$*message*. Stylistically, this form is useful for stateless services for which the identity of the recipient cannot matter. The run-time system is free to select any instance of the indicated class (or any subclass thereof) to receive the message. ODL does not prescribe a particular translation mechanism. Several are available. For example, because all occurrences of $ are statically determinable before execution, the system may construct a pool of such objects upon initialization and translate all invocations to normal prefixed form.

4.1.2 Procedures

In the base syntax of procedural interaction, result-bearing operations define message records for returned values, and clients define corresponding operations to accept them:

```
class A op m1(x: T) : result(b: B) ... end end;

class Auser ...
  op calla(a: A) {
    catch a.m1(x)
      op result(b: B) { b.m2(y, z); ... }
    end  }
end
```

Here, the sender enters a state in which its only next action is to receive a corresponding reply. A `catch` clause introduces one or more transiently available operations that accept replies from servers. Multiple named result messages and catches are also allowed. The names of the client operations must match those listed for the server. Translators must arrange that result operations be caught only when objects are in the required state.

A server object invokes these transient messages by name:

```
class A op m1(x: T) : result(b: B) { ...; result(new B); } end;
```

The recipient of the reply is left implicit. This logically requires that sender identity be transmitted as an implicit argument in all procedural messages. However, in ODL the sender identity is not otherwise accessible to the server. (Unless, of course, a sender field is explicitly added to the operation signature and used in the desired ways.) A server may perform additional actions after issuing a reply.

More conventional looking procedural forms are also supported, via via anonymous return messages. For example:

```
  op a : i: int;
  op b : ()
```

This declares the result of `a` as an anonymous record with single field `i` and the result of `b` as an anonymous, fieldless record. Each anonymous record is considered to have a different name. Anonymous return messages are sent via `reply`:

```
class A2 op m1(x: T) : B {  ...; reply b; } end;

class Auser  ...
  op calla2(a: A2) { local b :B := a.m1(x); b.m2(y, z);  }
end
```

Here, the anonymous `catch` may be elided, and the reply used directly in a procedural fashion.

Simple blocking procedural interaction is the only two-way protocol natively supported in ODL. Others may be defined through combinations of one-way sends and object constructions. For example, a *future* may be defined via the construction of a helper object to wait out a procedure.

4.1.3 Functions

Functional operations have a restricted form. They are defined as single expressions using the value expression sublanguage described in section 4.2.3. Translators are required to transform functional expressions into procedural computations (possibly involving new independent, unreachable objects) that cannot interfere with other operations and objects.

Stored functions are yet further restricted. They may be defined only via `<>`, indicating that a stored value be attached upon construction, retrieved upon access, and possibly rebound in the course of other operations. Stored links may also be qualified as `packed`. This is a hint to the translator that the object referenced by the slot should be embedded within the representation of the host object.

The base form of stored values is restricted to link values, not other types. The reflects an underlying object model in which state varies only as a function of connections among objects. Other forms may be implemented with the help of instances of elementary predefined classes. However, ODL programmers are not required to do so themselves. Translators may mechanically reduce them to base form. A stored value denoting a non-link type may be translated to one holding a link to an instance of a predefined class, where value accesses are forwarded to these objects. Value rebindings may be translated either to link rebindings of new objects with the required initial values or to `set` operations on the exisiting objects, or any other technique, at the discretion of the translator. Bindings of the form `l := null` for `opt` slots are handled similarly.

4.1.4 Local Operations

ODL `local` operation invocations are not received as messages. They are sequentially executed in the course of performing other actions. To avoid the need for redundant declaration, a *local* version of each functional non-local operation is automatically constructed if not otherwise present. Local operations must be invoked without a recipient prefix. (In contrast non-local self-invocations must be prefixed with `self`.)

Local functions and procedures may in turn invoke others, and may be recursive. Standard procedural invocation rules and semantics apply. One-way `local` operations are also allowed. Invocations are interpreted as structured "gotos" in which control does not return to the calling operation. For this reason, procedural operations may not invoke `local` one-ways.

The execution state of objects is in general unbounded. The existence and value of representational bounds for particular classes and objects may be conservatively assessed via static analysis of `local` operations. When bounds are not discerned or discernable, ODL implementations may establish maximal per-object run-time size limits and handle overflow as a run-time error.

4.1.5 Construction

Every instantiable class declaration automatically results in the definition of a corresponding `new` operation in class `System`. The `new` operation has arguments corresponding to all slots declared as `<>`, and returns a `unique` link value referencing an object of the indicated class. Implementations of `new` (as well as `delete`) are not definable within ODL, although provision of implementation-specific `blob`-based classes and object layout rules may make them so.

Without qualification, the class's `new` operation may be invoked anywhere. The visibility of `new` may be controlled via a `generator` clause in a class declaration. A `generator` clause names the classes of entities that may invoke `new` for the class.

4.1.6 Destruction

ODL message passing rules assume preservation of referential integrity. Objects that may still receive messages may not be deleted. This is best implemented using automatic storage management (garbage collection). However, a `delete` operation is also associated with each concrete class. Visibility is also controlled via `generator`.

4.2 Constraints

Constraints list properties of instances without otherwise defining concrete forms of slots. Any class may include any combination of constraints and concrete definitions for any slot. A class describing constraints but leaving one or more slots otherwise undefined is termed *abstract*. Abstract classes are not instantiable.

ODL constraints are *partial* descriptions. Objects obey declarative constraints, but any behavior that is not ruled out is possible. The forms of constraints are limited to those that may be evaluated via combinations of static symbolic analysis, translator assisted instrumentation, and dynamic checks.

Moreover, translators are only required to perform a subset of these measures, as described below. ODL does not specify whether or how conformance to others is enforced. ODL rules represent a compromise between expressive power and inferential and run-time requirements upon implementations. Reification of specification constructs renders ODL at best incomplete as a declarative language. However, common constraints remain simply expressible and checkable.

4.2.1 Bindings

In ODL, the bindings from names to all slots except stored functions are fixed and common to all direct instances of a class. (This restriction may be lifted in a future version.) Bindings for stored slots may be changed during execution (via `:=`) unless they have been qualified as `fixed`. The binding for a slot qualified as `fixed` remains constant across the lifetime of each instance. The qualifier `fixed` may also be applied to a non-stored function to indicate that its value does not vary over time. (Note in this case that `fixed` refers to the value, not the binding.) The keyword `own` is an abbreviation for `local fixed unique`.

4.2.2 Types

All arguments and results (including function values) for all slots must be constrained by type, and optionally by qualifiers `unique`, `common`, and `opt`. Annotations for link types are partial specifications – they list *a* type for the link, not necessarily the *maximal* (most exact) type.

In ODL, a message may be sent only if the recipient will eventually accept it. Determining conformance with this rule is in general undecidable. However, ODL programs must obey the weaker rule that a message be listed in a concrete definition only if it is provable that the recipient does not forever `pend` the message; i.e., if the message triggers a corresponding operation in at least one condition in a class declaration. The proof method is conservative, and based on type checking. In ODL every link is qualified with a type annotation indicating a class to which the referenced object must belong. The ODL type checker treats any attempt to send a message not listed in the indicated class (or superclass) as "not provable", thus as a programming error. However, the checker also admits invocations nested within conditionals checking (via "`in`") that the recipient is of a class supporting an operation. For example:

```
class A op m ... end end;
op calla(a: Any) { if a in A then a.m end; }
```

4.2.3 Function Value Constraints

Additional constraints may be declared via:

- Invariant (`inv`) expressions that hold whenever objects are not engaged in operations (i.e., at all quiescent states).

- Short forms of equality invariants for functional slots: `fn f` ...`=` *exp*.
- Initial condition (`init`) expressions holding upon construction.
- Short forms of equality-based initial conditions: `fn f` ...`init=` *exp*.

All constraints and conditions must be expressed within the (executable) value expression sublanguage of ODL. Invariants and initial conditions are boolean-valued expressions constructed from:

- Literals of value types.
- Equality operators (`=, ~=`) on values of all types.
- Relational ordering operators (`<, <=, >, >=`) on values of `int, char, real, time` types. (ASCII compatible ordering is required of `char`.)
- Operators defined on boolean values: `/\` (*and*), `\/` (*or*), `~` (*not*), `=>` (*implies*), and comma (`,`) (low precedence *and*). *And* and *or* are "short circuiting": successive terms of expressions need not be well-defined if the truth value is determined by previous terms.
- Operators defined on integer values (`+, -, *, div, mod`).
- Operators defined on real values (`+, -, *, /`).
- Real-valued mixed mode operators between integers and reals (`+, -, *, /`).
- Operators defined on time values (`+, -`), plus mixed mode `*, div` operations with integers.
- `if` *exp* `then` *exp* ...`else` *exp* `end`.
- Operator `null(`*link*`)`, defined on link types.
- Operator *link* `in` *class*, that is true if the object referenced by *link* is an instance of *class* or a subclass thereof.
- Field and subscript selection on records and arrays.
- Invocations of functional operations.

This sublanguage may be viewed as a very small pure functional language. ODL restricts the forms of functional operations in order to enable their use in declarative constraints. They thus play a dual role. From a computational perspective, they are restricted forms of operations, but from a declarative perspective they serve as symbolically tractable functions. The requirement that functional expressions be translatable into particular executable forms limits power and restricts expression. For example, bounded universal quantification is expressed via type annotations for function arguments.

In fact, `inv` and `init` are treated by translators as special declaration forms of ordinary functional slots. All `inv` constraints for a class are collected (clausally conjoined in listed order) in executable form as callable `fn inv: bool` that may be invoked at run-time. Similarly for `fn init: bool`. ODL does not otherwise require translators to perform symbolic analysis on constraints (e.g., to determine whether they are even satisfiable). Design checkers that perform such analyses may be constructed, but these capabilities are not demanded of translators.

4.2.4 Operation Constraints

Guards. Guards are "active" preconditions listing the conditions under which nonfunctional, nonlocal operations may be executed. ("Passive" preconditions describing alternatives within accepted operations are listed instead as `ifs` within effect expressions.) There are both "outer" and "inner" guards, of the form:

```
class C ...
 when c1 then
   op m1
     when m1c1 ==> ...
     elsewhen m1c2 ==> ...
     else ...  end
   op m2 ...
 elsewhen c2 then
   op m1 ...
 else ...  end
end
```

Any combination of "outer" clauses with embedded operations, and "inner" guards nested within operations are permitted. Stylistically, outer guards refer to object state, while inner forms refer to properties of message arguments of `ops`. Nested sets of guards are also permitted. Different *versions* of the same `op` may be declared in different arms of outer guards (as seen above for `m1`. Consistency rules for multiple versions are described in section 5.

Boolean expressions within `when` clauses are constructed using the above expression sublanguage. Translators must provide interference-free translation of guard expressions into executable form.

Like `elsifs` in most languages, the condition in each `elsewhen` clause is interpreted to include the negation of all preceeding conditions. This guarantees mutual exclusion of conditions. For example, `elsewhen c2` above is interpreted as `when ~c1 /\ c2`.

Pending. `Pend` is a pseudo-effect indicating that a message does not trigger an operation at all under the listed condition. If any other condition ever becomes true, the corresponding operation will be triggered accordingly. All explicit guards and non-`local` operations are considered to be encased within:

```
class ...
 when ready
   ...
 else
   pend
 end
end
```

`Ready` is true when an object is not otherwise engaged in an operation. (This function does not actually exist and cannot be expressed in ODL.)

By default, if an `op` is listed in only one outer `when` then it it assumed to `pend` in all others in which it is not listed. Completely unlisted messages `pend` forever. Functional operations may `pend` only when an object is busy in another operation. Local operations are not processed as messages and cannot `pend`. Thus, no explict guards may be associated with functional and `local` operations.

Translators may employ any non-interfering mechanism to implement `pend`. The use of guards does not require that translators establish identifiable per-object message queues. No ODL constructs refer to queues. No run-time support is needed if a translator can determine that no `pend` conditions can ever be encountered for an object. When any of several messages may be accepted (i.e., clear guard conditions), any one of them may be chosen. Implementations may provide stronger ordering guarantees. Implementation limits in the number of possible simultaneously pending messages are permitted.

Translators cannot always statically detect situations in which conditionally accepted messages forever `pend`. They may provide run-time mechanisms to assist users in dealing with resulting deadlocks and overflows.

Effects. Effects (`==>`) list conditions that must hold as a result of particular operations. Clauses within effect descriptions are defined using the constraint sublanguage extended with constructs:

`exp'` The value of an expression as evaluated upon completion of the operation. (Unprimed forms within effects refer to evaluation upon commencement of operations.)

`msg'` Assertion that the effects of `msg` hold at completion of the operation.

`msg''` Assertion that the effects of `msg` hold (perhaps only briefly) at some point after commencement of the operation.

`@boolexp` The time after which the operation commences at which the expression becomes true.

No translation into executable form is specified for effect expressions. In particular, effect expressions with double-primes cannot be translated in any useful manner since the time at which they should hold true is unbounded. However, static analysis tools may be constructed to determine satisfiablity of effect expressions and partial conformance of concrete forms. Other tools may be devised to help instrument checks for certain effects and/or to help generate test code.

`ODL` effects are *not* subject to "frame assumptions" that claim that properties that are not mentioned do not change. Any behavior consistent with constraints is allowed. All properties that must be preserved within `op`s should be explicit unless they are also listed in `inv`s (which serve as implicit pre- and postconditions for all operations). In contrast, functional operations do obey frame axioms since they are computed in a side-effect-free manner.

5 Inheritance

A class may list any number of superclasses. If none are listed, the class is taken to be a subclass of `Any`. `Any` is the root of the class system. It defines only the fixed slot `self`, providing a reference to self. Subclass declarations extend the scopes of their superclasses. The declared class structure determines the type structure for links. The type of a link referencing instances of a class is a subtype of superclass link types.

Subclass declarations extend those of their superclasses. All stated properties (slot definitions and constraints) in superclasses are preserved in subclasses. Additions may not invalidate superclass properties. Detected conflicts may be trapped as programming errors by a translator, but it is not specified whether or how conformance is enforced.

Subclass declarations add new slots and add (strengthen) features of those listed in superclasses. Additions are conjoined to the corresponding superclass declarations. Full redeclarations are unnecessary except in those cases where features cannot otherwise be expressed (e.g., when adding qualifiers). When a new slot has the same name as one in the superclass, or when two or more superclasses have slots of the same name, the following rules apply:

Multiple Declaration. Two declarations with the same slot type (operation, procedure, function, local), number of arguments, argument types and qualifiers denote the same slot. All other aspects of the declarations are coalesced as one.

Versioning. Two non-local, non-functional declarations differing in the declared *link* type of one or more argument fields, or differing in outer `when` guards are considered to be variant *versions* of the same slot.

Overloading. Two declarations differing in number of arguments, or in the types of one or more argument in the case where those types are incommensurate (e.g., `int` versus `real`, any non-link type versus a link type), or where only one is declared as `local`, are taken to be two unrelated slots that may coexist (i.e., as a case of *ad hoc* overloading). Variants with and without `opt` qualifiers on one or more arguments are similarly treated as overloaded.

Unsupported. Two declarations differing in any other way (e.g., `fn` versus `op`, qualifiers) are disallowed because invocation forms of the different cases cannot be distinquished in ODL.

The first two cases are instances of adding features to existing slots. The following additions are permitted. Analogous rules apply when merging the declarations of two or more superclasses.

- Adding a concrete definition to an existing slot.
- Adding a new clause to an existing effect.
- Adding a nonconflicting `inv` constraint. All `inv` clauses in the class and superclasses are interpreted as a single conjoined expression (with subclass clauses prepended to superclass clauses) that must be satisfiable.
- Adding a nonconflicting `init` constraint.
- Adding a constraint to a functional slot. The type of a function may be strengthened by replacing the result type declared in the superclass with a subtype thereof, adding `common`, `unique` and/or `fixed`, or removing `opt`. However, these may not conflict with other superclass constraints or definitions. For example, it is illegal to add a `fixed` qualifier if a `fn` value varies within a superclass operation.
- Adding a constraint to a procedure result. Result types and qualifiers of anonymous replies may be added in the same manner as for functions. Also, a constraint that one or more alternate named replies are not issued may be indicated by omitting them in the redeclaration. (The converse case of adding alternate named replies is not allowed.)
- Adding or subdividing an outer `when` clause into two or more subconditions in order to add a new operation or another version of an operation under one or more of them.
- Adding or subdividing an inner `when` clause into two or more subconditions in order to add a new clause to an effect and/or add a concrete definition under one or more of them. Effects and result types of all versions must meet all applicable superclass constraints.

These rules apply whenever a subclass adds a new version of a slot or two superclass versions exist. All declarations of different versions are interpreted as if they were different arms of a single operation with multiple `when` guards. A translator may fabricate guards (and/or perform equivalent transformations) in any way consistent with the declarations. For example, in:

```
class A ... end
class B is A ... end
class C ... end;

class D
  op m(a: A) ==> ea end;
  op m(b: B) ==> eb end;
  op m(c: C) ==> ec end;
end
```

The declarations of `m` in `D` may be transformed into a single operation, with all argument types recast in terms of their nearest common superclasses (here, just `Any`):

```
  op m(x: Any)
    when      x in B ==> eb
    elsewhen  x in A ==> ea
    elsewhen  x in C ==> ec
    else pend  end
```

The implicit negation of preceding guards in `when` clauses transforms consistency issues to ordering issues in the equivalent ODL code. For example, this may also be transformed as:

```
  op m(x: Any)
    when      x in C ==> ec
    elsewhen  x in B ==> eb
    elsewhen  x in A ==> ea
    else pend  end
```

In both cases, the version for `B` specializes that for `A`. However, class `C` bears no subclass or superclass relation to the others, so the clause may be considered in either fashion, at the discretion of the translator. For example, if the operation were invoked with an argument of type `AC` (a subclass of both `A` and `C`) then either version might trigger. While not disallowed, such non-determinism should be avoided.

The results of different versions of procedural operations may also vary. The combination rules are the same as would apply if each reply were considered as an operation proper. The base version is constructed by conjoining all cases as multiple replies, while merging (as the nearest common superclass) the types of identically named (or nameless) replies in those cases where fields differ only in link type. For example:

```
class E
  op p(a: A) f: A ;
  op p(b: B) g: B ;
  op p(c: C) h: int;
  op p(d: D)  ok(), bad(i: int)
end
```

This may be represented in the form:

```
class E
  op p(a: Any) fg: A, h: int, ok(), bad(i: int);
end
```

When a superclass version of the procedure exists, the resulting return expression must conform. In particular, added named reply forms resulting from such combinations are not allowed.

5.1 Subclass Restrictions

A constraint of the form `C = oneOf(S1, S2, ...  Sn)` limits the declarable subclasses of `C` to those listed under `OneOf`. Stylistically, `OneOf` is used to indicate that the listed subclasses are the only ones logically possible. For example, a class with a `fixed bool` slot might be partitioned into two subclasses with it set to true versus false. Also, subclasses defined via `OneOf` serve as analogs of enumeration types found in other languages. A translator may enforce and exploit the facts that partitioning constraints place fixed bounds on the number of subclasses and/or that partitioned siblings may never have a common subclass.

5.2 Defeasible Inheritance

A class may list another "base" class in an `opens` clause to indicate that it shares all but specifically redeclared features with this base. The rules are the same as those under normal inheritance except that *any* feature may be redeclared in *any* way – declarative constraints in the base declaration are ignored. Additionally, any unguarded base slot may be redeclared as `local`. The class listed in `opens` is not a superclass; link types of the derived class are not subtypes of those for the base.

6 Parameterization

needs revision

Classes, operations, and records may be paramaterized with one or more class arguments in brackets appended to the declared name. They may may then appear anywhere an ordinary type could appear inside the declaration. A parameterized entity simultaneously defines all possible specializations of that entity. The specializations are themselves ordinary classes, operations, and records, subject to normal use. Parameterized classes may be defined as subclasses of other parameterized classes.

Translators must generate specializations for all versions that are actually used in a program. They may generate others as well; for example, those used in possible but untaken computation paths. Because they remain controversial, ODL does not support *kinds* (types of types). There are no type constraints on type arguments. However, translators must detect and report errors when expansions result in specializations that contain (nonparameterized) type errors. Since instantiation is static, first-order type rules are maintained for all specializations used in a program.

7 Predefined Classes

Besides the special classes `Any` and `System`, the following abstract classes are predefined.

```
class Bool
  fn val: bool;
  op t!: ()   ==> val' = true end
  op f!: ()   ==> val' = false end
  op set(b: bool): () ==> val' = b end
end

class Char
  fn val: char;
  op set(c: char): () ==> val' = c end
end

class Int
  fn val: int;
  op inc: ()          ==> val' = val + 1   end
  op dec: ()          ==> val' = val - 1   end
  op neg: ()          ==> val' = -val      end
  op set(i: int):  () ==> val' = i         end
  op add(i: int):  () ==> val' = val + i   end
  op sub(i: int):  () ==> val' = val - i   end
  op mul(i: int):  () ==> val' = val * i   end
  op dvd(i: int):  () ==> val' = val div i end
  op rem(i: int):  () ==> val' = val mod i end
end

class Real
  fn val: real;
  op neg: ()          ==> val' = -val      end
  op set(r: real): () ==> val' = r         end
  op add(r: real): () ==> val' = val + r   end
  op sub(r: real): () ==> val' = val - r   end
  op mul(r: real): () ==> val' = val * r   end
  op dvd(r: real): () ==> val' = val / r   end

  op set(i: int):  () ==> val' = i         end
  op add(i: int):  () ==> val' = val + i   end
  op sub(i: int):  () ==> val' = val - i   end
  op mul(i: int):  () ==> val' = val * i   end
  op dvd(i: int):  () ==> val' = val / i   end
end

class Time
  fn val: time;
  op set(t: time): () ==> val' = t         end
  op add(t: time): () ==> val' = val + t   end
  op sub(t: time): () ==> val' = val - t   end
  op mul(i: int):  () ==> val' = val * i   end
  op dvd(i: int):  () ==> val' = val div i end
end
```

As syntactic sugar, the `val fn` for an instance of any predefined class may be accessed via postfix `?`.

At least one primitively implemented concrete subclass must be available for each of the above; minimally:

```
class BOOL is Bool ... end
class INT  is Int  ... end
class REAL is Real ... end
class CHAR is Char ... end
class TIME is Time ... end
```

There are no predefined classes supporting `blob` values. Implementations may provide them.

The partially predefined class `System` and single instance `system` are handled differently than all others. There need not be a single identifiable `system` object during execution. Its functionality may be spread (perhaps redundantly) across all processes or otherwise achieved in a constructed system.

8 Syntax

This still needs updating!

The following EBNF syntax uses “[...]” for “optional” and “[...]*” for “zero or more”.

System:	[Decl]*
Decl:	Class | Fn | Op | Inv | Init | Open | Gen | Locals | Accept | Rec | ;
Class:	class GID [is GIDs] [Decl]* end
Fn:	[local | own | packed] [fn] GID Params : QualType FnDef
Op:	[local] op GID Params ReturnSpec OpDef
Inv:	inv Exps
Init:	init Exps
Open:	opens GID
Gen:	generator GID
Rec:	record GID Params
Locals:	locals [Decl]* end
Accept:	when Exp then [Op]* ElseAccepts end
ElseAccepts:	[elsewhen Exp then [Op]*]* else [Op]*
Params:	[(ParamList)]
ParamList:	GID : QualType [, GID : QualType]*
QualType:	[fixed | unique | common | opt]* GID
ReturnSpec:	[[ID] : QualType | : Synch [, Synch]*]
Synch:	[ID] ([ParamList])
FnDef:	[[init] = Exp] FnBind
FnBind:	<> | Block | ;
OpDef:	Block | Effect | ;
Effect:	==> OpSpec end | When
When:	when Exp then OpSpec ElseWhens end
ElseWhens:	[elsewhen Exp then OpSpec]* else OpSpec
OpSpec:	[When | Exps [Block] | Block]
Block:	{ Statements }
Statements:	Statement [; Statement]*
Statement:	[Exp | Assign | Loc | Catch | While | If | Reply]
Reply:	reply [Exp]
While:	while Exp do Statements end
If:	if Exp then Statements ElsIfs end
ElsIfs:	[elsif Exp then Statements]* [else Statements]
Catch:	catch Exp [Op]* end
Assigns:	Assign [, Assign]*
Assign:	GID := Exp
Loc:	local GID : QualType [:= Exp]
Exps:	Exp | Exp , Exps
Exp:	[@] Exp2

Exp2:	[Exp2 OrOp] Exp3
OrOp:	\/ | =>
Exp3:	[Exp3 /\ ] Exp4
Exp4:	[Exp5 RelOp] Exp5
RelOp:	= | < | > | ~= | >= | <=
Exp5:	[Exp5 AddOp] Exp6
AddOp:	+ | -
Exp6:	[Exp6 MulOp] Exp7
MulOp:	* | / | div | mod
Exp7:	[Unop]* Exp8
Unop:	- | ~
Exp8:	PredefFn | PredefExp | Msg | (Exp)
PredefFn:	Msg in GID | null (Msg) | oneOf (GIDs)
PredefExp:	true | false | null | pend | literal
Msg:	Rcvr [. Send]* [' | '' | ?]
Rcvr:	self | [GID $] Send | new GID [([Assigns|Exp])]
Send:	GID [([Exps])]
GID:	ID | GID [Exps] | PredefType
PredefType:	bool | int | char | real | time | blob | Any | System
GIDs:	GID [, GID]*
ID:	[ID ::]* name