# *ADOT Objects:*
# *Units of meaning in the world of rigging*

*July 1, 2022*

T.R.I.E.
Adam Gravitis[a] D.R. Toliver[a]

[a]TODAQ

## *Introduction*

An ADOT object, or *abject*, is an object that can be embedded in rigging.

The encoding of abjects makes use of the atomic serialization protocol described in the rigging specification[1]. In particular, an abject is a list of atoms (or *lat*), with the final atom being the *focus* of the abject: a specific atom that uniquely identifies that particular abject, and with which that abject is uniquely identified.

[1] Coward, Toliver, et al., *Rigging Specifications*, 2022.

An abject is represented by a trie, and the focus atom is either that trie or a twist containing that trie as its cargo, depending on the abject. This trie must have a value for its NULL key[2]. The value of the NULL key in that trie is the *class* of the abject.

[2] NULL is the single zero byte `0x00`, and constitutes both a hash and an atom.

Each class has an *interpreter*, which takes an abject as input and produces a view of the abject as output. This *view object* provides natural and ergonomic interactions[3] within the host application.

[3] Something for which raw hash-based data structures are not generally known.

Producing a view object is contingent on the list of atoms satisfying the interpreter's demands, which may include checking various rigs as well as semantic considerations over the content of cargo tries. Insufficient proof data will produce an incomplete or invalid view object; malformed content will produce an erroneous one (see Section ).

Multiple abjects may make use of the same piece of rigging, though an individual twist may only contain one cargo trie, and therefore may only associate with a single abject.

An abject can be as simple as a static boxed value: the number 42, for instance. They can also be quite complex, defining actions and using those to maintain invariants across a wide range of stateful operations, with correspondingly complex rigging structures needed to prove that those operations have been performed and the invariants have been maintained.

## *Abjects*

An abject is a list of atoms. One of those atoms, the *focus* of the abject, is special: it defines the class of the abject, and is fed into the

interpreter of that class, along with the rest of the atoms, to produce the view object.

The focus must either be a twist or a trie, as specified by the class. If the focus is a twist then this is a *twisted abject*, and its fields are the fields in the twist's cargo. This is true in general: whenever a twist is supplied, its cargo trie is used as the source of fields for the abject. If the focus is a trie then this is a *tried abject*[4].

By convention, when serialized on disk, the focus is the final atom in the lat. This may change in other formats: a wire-level protocol may send the focus first, for instance.

[4] Pronounced 'treed', like 'retrieval'.

An abject's focus is a specific atom that uniquely identifies that abject, and with which that abject is uniquely identified. Any view object for that abject, regardless of the list of atoms provided, will agree with any other view object produced, as described in Section .

This unique association means that generally we can speak interchangeably about an abject and the focus of that abject. Likewise, because of the unique association between a twist and its cargo trie, we can speak directly about an abject's fields, without needing to dereference further.

### *Classes*

EVERY ABJECT HAS A SINGLE *class*, stored in the NULL field of its focus trie.

The class defines the rules to which that abject is subject. In particular, it must:

- Declare its focus shape (twist or trie)
- Introduce its symbols

The focus of the abject is either its trie, for abjects with no rigging, or a twist containing that trie as its cargo, for abjects whose interpreters do require rigging.

The symbols introduced by a class are typically unique, being freshly created for that class. Reusing symbols is discouraged, outside of a narrow set of exceptions. There is a tradeoff between being able to reuse the rels supplied for an existing symbol, which give it properties like its name and description in different languages, against the fact that these properties may well need to be slightly different in different classes, even for a seemingly very similar field.

### *Interpreters*

AN *interpreter* ACCEPTS AN ABJECT AS INPUT, and produces a view object as output. Each abject class has an interpreter, which must be available in the runtime client to process abjects of that class.

The job of an interpreter is two-fold. The first is to create a view object from an abject. The second is to create an abject by combining an existing abject with input data.

The view object that a given interpreter produces is spelled out in some detail below, but is not entirely prescribed. This is because an interpreter in language A may produce a different view object than the same interpreter in language B, given the same abject. For instance, given an INT32 abject, if language A has floats but does not have 32-bit integers as a primitive, then the INT32 interpreter in language A may produce a float[5].

[5] There is no INT32 interpreter. Mostly because various languages don't have 32-bit integers as a primitive, and keep producing floats instead.

Note that while the view objects produced by the same interpreter in two different environments may differ, the abjects they produce must be identical. The INT32 abject that language A's boxed value interpreter produces for the number 1729 must be identical to the INT32 abject that the language B's boxed value interpreter produces for 1729. For more complex abjects which include operations, interpreters may express those operations in whatever form is most suitable for that environment: procedures, functions, methods, typeclasses, data structures, interrupts, or whathaveyou.

### *View objects*

View objects provide a complete description of an abject within a given system.

A view object can be a primitive in the host language, like a string or integer for a boxed value. It can also be a more complex object, including having properties and methods for interacting with the abject, which the interpreter creates in the view object and fulfills.

Those methods can also potentially be used for constructing a fresh abject from this interpreter, when called with an empty input to produce an empty view object.

The values of the properties of a view object are often themselves view objects, so view objects are derived recursively as each interpreter hands off its view object to other interpreters and integrates the result into its own view object.

Each view object has one of three possible status choices:

- **valid**: The provided proof data was completely satisfying
- **incomplete**: The provided proof data was missing required information
- **invalid**: The required proof data is provably invalid

A valid view object does not necessarily need to show its status. In particular, any primitive produced as a view object must be valid.

A given interpreter may produce an object, error, exception, or some other device, in keeping with the language and runtime it is written within, in order to reveal information about incomplete or invalid abjects.

### *Errors*

View object status (valid, incomplete, invalid) reflects the state of the proof data: whether the rigging is satisfying, missing, or provably wrong. This is orthogonal to the content of the abject.

A view object whose content is malformed is *erroneous*. An abject can have valid proof status while its content is erroneous, or incomplete proof status while its content is well-formed.

An erroneous view object is treated as null within its parent context. This does not cascade: the parent sees null, not an error. Erroneous view objects must not be used for application logic, but may be delivered to the runtime for debugging purposes. Depending on the application, an erroneous view object may be ignored as a noop or may generate an application-level error.

Specific error conditions are defined by each interpreter. Common cases include:

- A boxed value whose arb does not match its type's requirements (e.g., invalid UTF-8, non-finite IEEE-754)
- A DI with a required field missing, a field whose value does not match its declared type, or a list field with a non-list value

### *Line traversal*

When a twisted abject's interpreter walks its line, it skips any twist that does not belong to it. Specifically, a twist is skipped if:

- It has null cargo
- Its cargo trie declares a different interpreter

Skipped twists are invisible to the interpreter: they do not contribute to the abject's view object, context, or JSON notes. This is not an error condition. A line may contain twists belonging to multiple abject classes, and each interpreter sees only its own.

Individual interpreters may define additional content-level skip conditions for twists that do carry their interpreter but lack expected content. These are specified per interpreter.

The rig covers the full line regardless of the interpreters carried by individual twists.

### *Atomic symbols*

This document makes heavy use of atomic symbols. Symbols are defined in the rigging specification as the hash algorithm byte $0x22$ followed by 256 bits of content, and form both a hash and a complete atom.

Symbols in this document are denoted by `0x22symbol`.

### *Terminology*

This document employs standard object/identifier punning, so the focus of an abject refers to both the focus atom as well as the hash of that atom, and the term abject refers to its list of atoms as well as the hash of the focus atom. Explicit disambiguation is provided where necessary.

## *Boxed values*

The first family of interpreters we define is for boxed values. They are simple wrappers, which are generally expressed as primitive values in the view object. The three boxed values are booleans, floats, and strings.

Each interpreter in this family shares the same symbol for its value. The value of the value in boxed values is always a value[6]. The focus is always a trie.

If the bytes within the arb do not match the requirement, the view object of that boxed value is erroneous (see Section ).

[6] I.e. the content of symbol 0x22882f...2b7095 is an arb packet.

**VALUE**

`0x22882f98c8f0396585a5a040258ee7af1fa658b12b323747207a634bfee62b7095`

### *Boolean*

**BOOLEAN**

`0x2209b7efb95a393d9ee01f6446f362e5cf56d5eee55b00e6118db367e28c6a945e`
Focus is a trie

The value of a Boolean abject is false if it is zero length, or the first byte is `0x00`. Any other value is true[7].

[7] Hence Booleans are never erroneous.

### *String*

**UTF-8**

`0x22afcfeed9b1c0a28ed7d197f23e7d33272bdb562aa8d9ccf151b8f9767ca09032`
Focus is a trie

Must be a proper UTF-8 string.

*Float*

**IEEE-754**

`0x22f2781cc0020c300c84c3b9bfbe38ceb8949f2c1ced61e29f25c90ff853eb83e4`
Focus is a trie

Must be a proper double-precision big-endian 64-bit binary format IEEE 754 finite number value. Note that both infinities and NaNs are explicitly excluded from this definition. Implementations must not produce those values in a view object.

*Examples*

```
△{
   0 : UTF8
   value : "some string arb packet"
  }

△{
   0 : IEEE-754
   value : 42
  }
```

## *Relationships*

The relationship interpreter, **R1**, defines a triple of entity, attribute, and value. This allows statements to be made about abjects *externally*, in addition to the internal statements made inside that abject about itself. Those internal statements only need two entries, an attribute and its value, because the identity of their target abject is uniquely determined by the statements themselves. Making a statement about a different abject requires that it be identified, requiring the third field.

External qualities are things like names. The human name for a given property will vary by language, region, speciality, and so on. Keywords in computer programs vary also, as different languages have different allowable character sets in their property names, and different idioms. It is important to have names, descriptions, icons, and other qualities be externally associated in an open environment, rather than internally defined and closed from expansion.

**R1** `0x220f6bc568a5f958111ce3c9d022bb03cf236827303b22b0af9d53eacf886c59ce`
R1's focus is a trie

R1 introduces three symbols that are used as keys. Its value field is distinct in purpose and symbol from M1 and boxed values.

**ENTITY**

`0x22310aca6f86516260b3520396c7c55fd395797347844bcd16f2901826e9ee9517`

**ATTR** `0x22e3cd5d34d6ecebf7be5d9922427d2d82f0fce97d4abb84d21c277c34f4692a25`

**VALUE**

`0x2252ddfdca28b6970128e5cdccdb373a2952c7183f572b97408293849d57801de8`

The following symbol is used by convention to attach a list of rels to an abject. The uses of this in production are outside the scope of this specification, but include applying names and descriptions in various languages, keywords appropriate for coding environments, alternate icons, and accessibility concerns.

**RELS**

`0x2297b3532254f96b588fdd7bfe7244ad03355d4de0764373354d7c6b65ac87886e`

*Examples*

Use rels to provide translations, regardless of the original source:

```
△{
   0 : R1
   entity : title-field
   attr   : en-US
   value  : △{
               0 : UTF8
               value : "Title"
              }
  }

r ← △{
   0 : R1
   entity : title-field
   attr   : fr-CA
   value  : △{
               0 : UTF8
               value : "Titrel"
              }
  }
```

Since rels are abjects, and all abjects are inherently named, we can use rels to provide metadata about other rels:

```
△{
   0 : R1
   entity : r
```

```
  attr   : error
  value  : bad-translation
 }
```

We can also fix the problematic translation just by including another rel:

```
△{
  0 : R1
  entity : title-field
  attr   : fr-CA
  value  : △{
             0 : UTF8
             value : "Titre"
           }
  }
```

## *Media types*

The **M1** interpreter allows existing content to be wrapped into an abject and unwrapped by the recipient. This interpreter is specifically for wrapping legacy content and system files.

**M1** `0x2285f216a38f53803b9c22de0f8fa335cae4c5c6a4faa28491d4ff2676867230c9`
M1's focus is a trie

This provides an integration pathway that emphasizes easiness over elegance, while still allowing interoperation with the other mechanisms of abjects, like relationships, actionables, and DI abjects. This can support a wide variety of use cases, such as:

- adding version control to any document or bundle of documents, including those with heavy branching or remixes;
- enforcing a single canonical versioning over a bundle of documents, like an M&A contract or a will;
- incorporating existing documents into a TODA file, like rights agreements in a product identifier.

The media type interpreter is not responsible for generating the view of a media type abject. Instead, it presents the host program with a standard media type view object, which contains the type-name and subtype-name fields, whose values are strings, and value field, whose value is a binary blob.

A host program should make a best effort attempt to generate a view of the media type, but should not compromise security,

performance, user experience, or general happiness of the host program developers. A host program can choose a set of media types it will process, or a set it will ignore, or decide dynamically whether to process. Host programs are under no obligation to make known their decisions about which media types they support.

Note that **M1** shares the value symbol of the boxed values family of interpreters.

**TYPE-NAME :: UTF8**

`0x228174bc1a34b9465332d984d18614627c0865a6db30dd74e4bcd6123dbf6ce7fb`

**SUBTYPE-NAME :: UTF8**

`0x221a48826ccde85fc944030e8c1abb651d4bd7fe8bf91497900b45da0d219ed864`

**VALUE :: ARB**

`0x22882f98c8f0396585a5a040258ee7af1fa658b12b323747207a634bfee62b7095`

### *Examples*

```
△{
   0: M1
   type-name: "application"
   subtype-name: "svg"
   value: arb
  }

△{
   0: M1
   type-name: "application"
   subtype-name: "midi"
   value: arb
  }
```

## *DI abjects*

Abjects in the **DI** interpreter express static content.

**DI** `0x22fcef42f4592bb500a6e03fdb0c80ef679e5dce3cbb3c1ab986108b86651ccb12`
DI's focus is a trie

DI abjects have a field template, which describes the fields an abject of that field template is expected to have. The field template is used by the DI interpreter in providing the view object for a DI abject, and can also be used by a user interface for building or editing a DI abject.

DI abjects are expressed as bare abject tries without any rigging, like boxed values and rels. They may be used as a twist's cargo,

but the DI interpreter ignores everything except the cargo trie, and its canonical representation excludes those extra atoms. DI abjects are often included inside other types of abjects, and serve as a basic building block for complex types.

DI abjects may contain information for humans, such as a description, or a logo[8]. They may also contain information for machines, such as a set of rules to follow, particularly when a DI abject is used within an actionable.

[8] Note that while the field names and descriptions are open and provided by rels, the values of particular piece of content are closed. Rels are then used to link two different pieces of content, claiming for instance that one is a translation of the other.

### *Field templates*

A DI abject may declare its *field template*, which is a DI abject that describes the fields its member abjects are expected to have. Fields are symbols introduced by the field template. Those symbols are used as keys in a trie.

The qualities described by a field template are the closed, internal qualities of that field, such as the type of the value it is expected to take. For qualities like names and descriptions rels should be used, as they provide an open, extensible interface.

The term *field* is used to refer to a field template definition ("required field"), the symbol ("all required fields must be present"), and the key-value pair in a trie ("the field's value"). Fields are called *properties* when they appear in a view object.

DI introduces the following symbols to support the field template system.

**FIELD-TEMPLATE :: ABJECT**

`0x22438bff6088ea812122de07b4329c5853f1ca9e2bfe8d65c2433c4514f3b8f837`

**FIELDS :: TRIE**

`0x22d29913f7eb9b76f0a1227d0b34465b7adf2236452e20734197e40da790f1f00d`

**REQUIRED :: BOOLEAN**

`0x22410489d7e5b4d32f75888c24eb20765342e670fc2616969cbb1fd06e3d3324d5`

Defaults to false. Note that booleans are boxed values.

**TYPE :: SYMBOL**

`0x22dba83636eaa2a14b9cc219669a4f82b7fe6d08cdd4318b5bfa37d51d47a9bf4f`

Defaults to unrestricted. Can be an interpreter, a field-template, or some other descriptor.

**LIST :: BOOLEAN**

`0x2299980d10e44dff83b24b80472098e40ff5fee15d70a3a5d2cfac5c47311929f5`

Defaults to false. If true, a list of entries is expected in a `0x61` shaped hashlist[9].

[9] An atomic shape defined in the rigging specification.

**CONSOLIDATION :: SYMBOL**

`0x225c499de98d731839873cd66e4d84532e53162328c377d1b7fc057630d03f0436`
See Section  for details.

A DI abject may contain fields which are not defined in its field template. These unlisted fields are treated in the view object just like listed fields that have none of the field template fields set (i.e., they have the default values). Any rels present should be applied to these fields as usual.

When creating a field template it is customary to set its field-template to the field-template symbol. This is the bootstrapping case: the standard field template fields (required, type, list, consolidation) are applied to the entries in its fields trie[10]. Any DI may be used as a field template if it has a value for its fields field. DI abjects that have no field template are merely treated as though all the fields present are unlisted, as above. DI abjects that have a non-DI field template (other than the field-template symbol), or whose DI field template has an inappropriate value or no value for its fields field, are treated as though they have no field template.

[10] This is the recursive base case. Hashes cannot contain themselves, so the field-template symbol serves as the termination signal.

None of the standard field template fields are themselves required. If the required or list fields have values that are not valid booleans, they are treated as false.

If a required field is not present then the view object of that entire DI abject is erroneous. If a field has a type specified, and has a value, and that value does not match its type, then its view object is erroneous. If a field is specified as a list, and has a value, and that value is not a list, then its view object is erroneous. An error in any field marks the view object of its entire DI as erroneous. See Section  for the general treatment of erroneous view objects.

*Examples*

Here **X** is a new DI abject, **FT** is its field template, and **FF** is the field definition trie inside FT. Note that FF is not an abject, it is merely a data trie. Even if it had its NULL key set, the DI interpreter still wouldn't treat it as an abject — but it does treat the value of the field-template field as an abject. Interpreters have a lot of latitude in their interpretation.

```
X ← △{
       0: DI
       field-template: FT
       names: ["Bob", "Dobbs"]
       phone: "1-555-231-3213"
      }
```

```
FT ← △{
        0: DI
        field-template: field-template
        fields: FF
       }

FF ← △{
        names: △{
                  type: UTF8
                  list: true
                 }
        phone: △{ type: UTF8 }
       }
```

A field template can have keys other than those listed above. This can be helpful for packaging content that describes the field template and its fields. In the example below a pair of rels are included that provide titles for the fields. The 'rels' key is not defined in field-template, but the DI interpreter is open: it attempts to put all available fields into the view object, including those that are not in the field template, on a best effort basis.

```
△{
   0: DI
   field-template: field-template
   fields: △{
              names: △{
                        type : UTF8
                        list : true
                       }
              phone: △{ type : UTF8 }
             }
   rels: [
           (names : title : "Full name")
           (phone : title : "Phone number")
         ]
}
```

*Consolidation*

Given a list of DI abjects, homogeneous over field template, we can consolidate that list into a single view object by using the consolidation strategy expressed for each field in the field template.

Consolidation provides a simple and consistent method for producing a view object from a homogeneous list of DI abjects. These lists are a building block for adding content to more complex abjects, such as actionables, which use rigging to maintain their invariants.

The initially supported consolidation strategies follow. Values are entries for that field. Abjects without that field cause no change to that field's value.

The default strategy is append. This is applied to fields defined in the field template that are lacking a consolidation entry.

Append is also used for keys that are not defined in that field template. This also includes keys that are not present in the first item in the list. The final consolidated view object will contain an entry for every field encountered in every DI abject in the list.

#### APPEND

`0x2295df977c3405f37820d6b03f54785c35beba127da5cc3d5ec442206d54656376`

Default strategy. Puts every value into a list.

#### REMOVE

`0x2238cb2d3d05737963c33c391a99b06a3db6d24bbfdc18a00f595d7ab0c386c6e7`

Takes matching values out of the initial list, reducing it in size.

#### LAST-WRITE-WINS

`0x221d98c3cedd4c0ff12458b2e22d270fcf6f45ca8ffaf64dd8eca957599d3ff562`

Takes only the most recent value.

#### FIRST-WRITE-WINS

`0x222276108951a0926d11418a8446b01051177d720227ec2a10bd57c1b4e261f4f3`

Takes only the first value.

The append strategy appends everything into the result list, with no further consolidation: a list value is appended to the result list, a trie value is appended to the result list, and so on. NULL values are not appended to the result list, whether they are explicit or implicit NULLs from the absence of a field in a DI.

The remove strategy requires each value to be a list. If the first value is not a list, the resulting list is always an empty list. If any subsequent value is not a list, it is ignored. If values are present in a subsequent list that are not in the first value's list, they are ignored. If multiple instances of a value are present in the first value's list, one instance is removed each time that value appears in a subsequent list.

Note that consolidation strategies themselves never produce erroneous view objects. However, because the remove strategy requires the values of a field thus marked to always be lists, it is reasonable to also specify it as a list in the field template. Then if the value is not a list, the view object of the entire DI will be erroneous,

and will show up as NULL in the list of DI abjects for consolidation, and be skipped.

### *Twisted abjects*

A twisted abject is any abject whose focus is a twist. In this version of the specification, all twisted abjects are actionables.

### *Actionables*

Actionables are a family of interpreters that share various traits. Actionables are twisted abjects whose interpreter expresses a single action.

All actionables are rigged. A new **poptop** symbol is introduced, shared by all actionable interpreters. Actionables are expected to maintain the same poptop for their entire lifecycle.

The focus of every actionable is a twist.

**POPTOP :: TWIST**

`0x22c70173874680c58e5c1d32854bd10486aac6f1aa821b56e3d512fd72e45ac72e`
required, unless delegate-initiate

Each actionable twist in a line uses its predecessor's poptop, back to the first actionable of that type in the line, skipping any intervening twists per Section . The actionable's lat must contain a rig proving support of the actionable by the poptop.

Some actionable interpreters provide delegation, and their actionables that are delegates use their delegator's poptop.

Each actionable interpreter decides what targets it allows for its action. It also decides on the view of its abject, based on the actions it has taken or that have been taken on it.

Actionables rely on their context, an optional DI abject, for semantically interesting content, like rules or claims. Extensible content like names and descriptions are provided by rels.

### *Context*

The context of an actionable is a self-selected description of the abject. Note that the binding here is one-way: the actionable points to its context, but the context does not point back at the actionable. It is a description, not a definition, and given an actionable it is trivial to create a second one with the same context.

The context of an actionable usually has important characteristics for that actionable, which are visible in the view of the actionable.

Some actionable interpreters may require an abject of a particular field template as their context.

It is common, but not mandatory, for actionable interpreters to use DI consolidation over their contexts. For actionables that delegate this consolidation may include contexts from the delegation chain in addition to their own line.

**CONTEXT**

```
0x2208318633b506017519e9b90b0bdc8451772415ba29144ab7778cb09cc2d2fa6a
```

### *JSON notes*

The JSON notes field provides a channel for app-specific content in any actionable. It is optional in each twist's cargo. Its value is an arb packet containing valid UTF-8 encoded JSON. If the bytes are not valid JSON, the field is treated as absent. An empty JSON object is a valid entry.

Some apps may ignore this field entirely. For content that should be interpreted consistently across applications, use the context field instead.

The view of JSON notes is a chronological list, oldest first, of all valid JSON entries from this field encountered during interpretation.

**JSON-NOTES :: JSON**

```
0x22a38c817efa88c1f2c6f290f4104387f1484e06476882690b0e99399d2879640f
```

### *Actions*

Actionables can use *actions*. Actions come in three different varieties. Each actionable interpreter introduces fresh symbols for its actions, which are used as keys in the actionable's cargo. The values of these keys are always abjects, or lists of abjects.

Each actionable interpreter chooses a type of action from the list below, and introduces new symbols for its verbs. Each verb takes an abject as its input.

- **One way actions** introduce a single symbol, a *cast*. These actions are used to give special properties to tried abjects, like DI, or to imbue another actionable with some value.
- **Two way actions** introduce two symbols, a *send* and a *receive*. The abject that contains this send symbol as a key is called the sender, and the abject that is the value of the send key is the recipient. The sender must be an actionable with this interpreter; the recipient

can be any twisted abject. The recipient must set the value of the receive key to the sender in one of its successors, and must prove it has done this with a rig to the sender's poptop, for this action to be successfully completed. Two way actions are also referred to as double binding.

- **Three way actions** reverse the ordering of a regular two way action: the "receiver" goes first, instead of the "sender". They introduce three symbols, an *initiate*, a *confirm* and a *complete*. An abject of this interpreter may initiate this action by setting the value of the initiate field to a twisted abject. That abject, the confirmer, must set the initiator as a value in the confirm field of one of its successors. This successor is called the confirmation. A successful confirmation is proven by providing a rig from the confirmer's line to the initiator's poptop's line, supporting the line between the twist referenced by the initiate field (i.e., the confirmer) and the confirmation. The initiator must then set the value of its complete field to the confirmation. The proof of this final step naturally requires a rig from the initiator's line to the initiator's poptop, spanning from the initial initiate to the completed complete. Only at this point is the action considered complete.

  Three way actions are used when an abject needs to receive something from a different abject, and prove that it has done so. The interpreter determines whether the confirmer must be the same type of abject as the initiator, as in delegating, or a specific different type. The interpreter may also require the entirety of the confirmer abject, instead of just a minimal portion of its line.

  The interpreter also determines whether the initiator must be the first twist in a line. This is the case in delegation, and it creates a very tight lifelong bond between the two abjects. This is called initiation binding. Most actions do not require such a tight bond.

### *Delegation*

Delegation shares something from one actionable with a fresh actionable of the same interpreter. Delegation is an optional mixin: each actionable interpreter chooses whether to allow delegation.

Delegation is expressed as a standard three way initiation action. It introduces three new symbols, corresponding to the steps of the three way action, initiate, confirm and complete.

**DELEGATE-INITIATE :: TWIST**

`0x22251dbe656f28f8fd46de35a13c1d74921cb73c1c198800b77eb2417f09435a82`
required if no poptop

#### DELEGATE-CONFIRM :: [TWIST]

```
0x2246de612f227162a3d60819c45d88ba2d88d74aa86d64f865bf371be5ec8c52f0
```

#### DELEGATE-COMPLETE :: TWIST

```
0x229b2a6d33408bc08d1af4ec63f0fb8e627d6e3b4d3f208e90390c3d8df789de34
```

The delegate must set its delegate-initiate field to the delegator, and must not set its poptop.[11] The delegate uses the poptop of its delegator. This applies transitively if its delegator is also a delegate.

The delegator must respond by adding the delegate in the value of its delegate-confirm field. The delegate-confirm field holds a list of delegates.

The immediate successor of the delegate-initiate twist must have a delegate-complete field in its cargo trie[12], and the value of that field must be the delegate-confirm twist. Until this occurs the delegate is in an incomplete state, and is not usable.

For actionables that can delegate, each abject is either a delegate or a root. The rig for a root abject is checked against its poptop as normal.

Each delegate abject must also provide a rig proving support from the root's poptop. This applies recursively to each delegate in the delegation chain.

For a delegate C, with delegation chain A, B, C, with A being its root, the following lines must be checked for support against A's poptop:

- A from first twist to its delegate-confirm of B
- B from delegate-initiate to its delegate-confirm of C
- C from delegate-initiate to the actionable's focus twist

Delegates may be limited in their authority by the semantics of the actionable interpreter.

Delegation is tree-shaped. If delegation is allowed, delegates may in turn delegate to new delegates, unless the interpreter restricts delegation depth.

Delegation is permanent. There is no way to revoke a delegate once it is approved. A delegate can be tethered to UNIT[13] as a convenient way of indicating it is no longer usable. Doing this does not affect any delegates of that delegate. The semantics of some actionable interpreters may allow delegates to be constrained in time, for instance by setting an expiry date.

[11] If the poptop is set, it must be ignored. Correct clients must not set the poptop. Incorrect clients are to be publicly shamed and mocked mercilessly until they comply.

[12] Which must be an abject trie with the same interpreter.

[13] UNIT is the atom consisting of the all-1 byte (255 in decimal). It serves as a universal sentinel value.

### *Simple rigged*

The **SR** simple rigged actionable interpreter provides integrity across updates to its content.

**SR** `0x224a77394f604847ace4358961d501d95c19ec9b9572ee877368a274411daf01fb`
SR's focus is a twist

This allows history of this abject to be supported by the full integrity of its poptop. Every other version of this abject must be compatible with it, up to poptop equivocation.

The simple rigged interpreter introduces no actions.

Simple rigged abjects cannot delegate.

Their view object is simply DI consolidation over the list of context abjects. If the focus twist has no context field, the context list is empty. When assembling this list, twists are skipped per the general line traversal rules (Section ). Additionally, twists that lack a value for the context field, or whose context is not a DI, or whose context DI's field template does not match the focus's context's field template (by hash identity), are also skipped.

### *Shallow delegable*

THE **SDA** SHALLOW DELEGABLE ACTIONABLE INTERPRETER provides a simple delegable abject without quantity. An SDA root can create delegates, but those delegates cannot delegate further, restricting the delegation tree to depth one.

**SDA** `0x224a77394f604847ace4358961d501d95c19ec9b9572ee877368a274411daf01fc`
SDA's focus is a twist

An SDA can delegate, subject to the restriction that delegates cannot delegate further. If a delegate attempts to initiate a delegation, it is erroneous.

The SDA interpreter introduces no new actions.

The SDA context field template:

**MINTING-INFO :: DI :: FWW**

`0x220b0bfdd07d701255a52dff626c4a69b7af73e42061857b4b04537542c4e4ea52`
A DI abject describing this SDA. This symbol is shared with DQ.

The view object of an SDA is derived from DI consolidation over the list of contexts in its delegation chain, starting from the root context. If the focus twist has no context field, the context list is

empty. Twists are skipped per the general line traversal rules (Section ), and additionally if they lack a context field, or their context is not a DI, or their context DI's field template does not match the focus's context's field template (by hash identity).

The view object contains:

- **minting-info** from the root context (first-write-wins)
- **root id**
- **delegation status** (root, incomplete delegate, or complete delegate)

The SDA class restricts delegation in the following ways:

- The **delegate-initiate** action must be in the first twist in a new line
- The value of **delegate-confirm** should be a list containing a single hash
- Only a single delegation action can be taken in a given twist
- Delegates cannot delegate: a delegate that attempts to initiate a delegation is erroneous

Applications commonly attach additional information to delegates using an app-specific field in the delegate's context (e.g., a delegate-info DI describing the purpose or destination of that particular delegate). This content is outside the scope of the SDA interpreter, but its presence is a widespread convention.

## *Quantity Invariant*

Abjects in **DQ** have a quantity. They provide a simple way to maintain that quantity through fractional delegation.

**DQ** `0x220a6a20be9131b708b193e1373aa4df209719e1d3f451836fa62245e4aed234a7`
DQ's focus is a twist

A DQ can delegate.
The DQ interpreter introduces no new actions.
The DQ context field template:

**QUANTITY :: FLOAT :: APPEND**

`0x229cd0e35e7f233a1c03f620f7c5024baf35c229df81ad613c622996bc1dc4da37`
A natural number[14]

[14] Enforced by interpreter; see below.

**DISPLAY-PRECISION :: FLOAT :: FWW**

`0x2248a88ced3cb2ee7e8187fccc4d70dad8ec75bb8f01b5dbfcdf94ef0ce4fcaea4`
A natural number

**MINTING-INFO :: DI :: FWW**

`0x220b0bfdd07d701255a52dff626c4a69b7af73e42061857b4b04537542c4e4ea52`
A DI abject describing this DQ

The view object of a DQ abject contains:

- **quantity**
- **display-precision**
- **root id**
- **minting-info** from the root context

A delegate $S$ can *peel off* an amount from its delegator $T$ by setting the quantity field in its context. This must be confirmed by $T$, and completed by $S$, before the delegation is complete[15].

The DQ class restricts delegation in the following ways:

- The **delegate-initiate** action must be in the first twist in a new line
- The value of **delegate-confirm** should be a list containing a single hash
- Only a single delegation action can be taken in a given twist
- If multiple potential delegation actions are present in a twist, the action taken is chosen by priority: first **delegate-initiate**, then **delegate-complete**, and finally **delegate-confirm**[16].

The DQ interpreter computes a single quantity value by walking the delegation chain and tracking how value flows through confirmations. The general algorithm is given below; the following rules illustrate it for a three-level chain. Here $T$ is the root, $S$ is a delegate of $T$, and $R$ is a delegate of $S$.

- The first twist of the root $T$ contains the full value of $T$ according to its quantity field.
- The first twist of $S$, which contains its delegate-initiate action, has no value.
- The twist of $T$ containing the delegate-confirm for $S$ has $S$'s value removed from $T$.
- The second twist of $S$, which contains its delegate-complete action, has the value of $S$.
- If the first twist of $R$ points at the delegate-initiate twist of $S$, then $R$ receives no value, because $S$ has no value at that twist.

[15] This is the standard delegation mechanism. See Section  for details.

[16] As a specific example, if a twist contains both delegate-confirm and delegate-complete, only the delegate-complete occurs.

- If the quantity of $S$ exceeds the quantity of $T$ then $S$ receives the value in $T$ and the value of $T$ is zero.

A valid quantity has a value that is:

- An integer
- Equal to or greater than zero
- Less than $2^{53}$

An invalid quantity is treated as having a value of 0.

The delegation action allows the delegator to confirm multiple delegates in a single twist, but this is not allowed in DQ. If there is more than one delegate listed in the delegate-confirm field then none of those delegates receive any amount, and the delegator retains its full amount. All of those delegates are permanently set to 0 quantity[17].

[17] This special treatment of the delegation action in DQ avoids the need for complicated resolution over simultaneous delegates. This is particularly true when the total amount delegated exceeds the amount available in the delegator, which may require detailed examination of the list structure of the delegate-confirm field (the delegate action generally treats it as a set) and increases the average proof weight requirements.

The following is the general procedure to calculate the quantity of a DQ abject $D$ with delegation chain $D_0, D_1, \ldots, D_n$ (where $D_0$ is the root and $D_n = D$). The *requested quantity* of a delegate is the quantity declared in the context of its delegate-initiate twist.

1. Set $value(D_0)$ to the quantity declared in the first twist of $D_0$. If this is not a valid quantity, $value(D_0) = 0$.
2. For each link in the chain ($D_i$ confirming $D_{i+1}$):
   (a) For each confirmation twist on $D_i$'s line before the one confirming $D_{i+1}$, in chronological order:
      - If it confirms exactly one delegate: let *claimed* be that delegate's requested quantity (0 if invalid). Set $value(D_i) = \max(value(D_i) - claimed, 0)$.
      - If it confirms multiple delegates: no deduction.

   (b) At the confirmation twist for $D_{i+1}$:
      - If it lists more than one delegate: $transferred = 0$.
      - Otherwise: let *claimed* be $D_{i+1}$'s requested quantity (0 if invalid). Set $transferred = \min(value(D_i), claimed)$. Set $value(D_i) = value(D_i) - transferred$.

   (c) $value(D_{i+1}) = transferred$, but only after $D_{i+1}$ completes. Until then, $value(D_{i+1}) = 0$.
3. The quantity of $D$ is $value(D_n)$.

Calculations using floats are unstable and cannot preserve numeric invariants. This procedure constrains the use of floats to small[18] natural numbers, where the quantity invariant can be preserved.

[18] Less than $2^{53}$, to be precise.

For DQ abjects where it is important to display fractional values, the display-precision field provides the interface with guidance on where to put the decimal point. A display-precision of 0 indicates that the user agent should show the quantity with no fractional portion: a quantity of 1234 should be displayed as 1234. With display-precision set to 3 the same quantity should be displayed as 1.234.

The display-precision field may take on integer values ranging from 0 to 15, inclusive. A DQ with an invalid display-precision field should be marked as erroneous.

## *Conclusion*

This specification defines abjects: objects embedded in TODA rigging that carry meaning, maintain invariants, and can be understood by any compliant implementation.

An abject is identified by its focus, classified by its NULL key, and interpreted to produce a view object. Tried abjects, like boxed values and DIs, are bare tries with no rigging. Twisted abjects have a twist as their focus, giving them a line and a history. All twisted abjects defined here are actionables: they share a poptop for integrity, and may use actions and delegation to interact with other abjects.

The specification defines three actionable interpreters. SR provides integrity over content that changes over time. SDA provides shallow delegation, creating single-level trees of delegates from a root. DQ provides fractional delegation with a quantity invariant, preserving value while it flows through arbitrary-depth delegation chains.

Several cross-cutting mechanisms support these interpreters. The error model separates proof status (valid, incomplete, invalid) from content status (erroneous), keeping the two concerns orthogonal. Line traversal rules allow multiple abject classes to coexist on a single line without interference. JSON notes provide an app-specific content channel alongside the structured context field. And consolidation strategies (append, remove, last-write-wins, first-write-wins) govern how field values accumulate over a list of DI abjects.

The mechanical operations of each interpreter — how quantity is computed, how delegation is verified, how integrity is checked — are frozen: they must produce identical results in every implementation, everywhere, forever.

The semantic content answers questions of meaning. It is layered on top of the mechanical operations, and is open and evolving. What an abject means, how it is displayed, what an application does with it can all be changed over time, without affecting the mechanical guarantees.

This separation is deliberate. The mechanical layer is unchangeable.

The semantic layer is adaptable. Like the paper assets on which they are based, abjects combine fixed rules with open modifications that are entirely in the hands of the possessor.