`ftillite`: Python FTIL-like API for software PoC

Michael Brand

November 21, 2023

Executive Summary

We describe a Python API that delivers to Python, substantially, the capabilities of FTIL, but without any of the FTIL safety mechanisms.

1 Introduction

As a Proof of Concept, the decision was made to enable the agility of FTIL (without the safeguards of FTIL) in a Python environment. This document defines a Python library, `ftillite`, that delivers this functionality.

This library was derived by means of considering the FTIL specifications, as described in [1], and then defining a minimal set of APIs that delivers this functionality. The library does not attempt to provide any of FTIL's safeguards, and does not optimise against unnecessary data duplication to the extent that FTIL does.

FTIL Note 1.1. This document includes some "FTIL Notes" such as this one, whose purpose is to draw attention to how `ftillite` functionality enables or maps onto FTIL functionality. These notes are not necessary for the implementation of `ftillite`, but explain the rationale behind some of the choices, and provide an outline for how `ftillite` can be used as a starting point for the implementation of FTIL in full.

2 Top level view

This section presents an overview of the library, its structure, how to work with it, and the function of individual elements. The main, complex classes are mentioned here, but are later given detailed analysis in their own, individual sections. The smaller, helper classes (`Node`, `NodeSet` and `Transmitter`) are introduced entirely here and not expanded on further.

2.1 Library design

The purpose of `ftillite` (and of FTIL) is to run privacy-preserving algorithms which are, necessarily, distributed. Specifically, we assume a small network of computers, all of which know each other in advance and all of which can communicate with each other directly.

Conceptually, for a Proof of Concept, the system is meant to simulate a situation where the network is composed of 5 computation nodes, running in parallel. In the simulation scenario, one node is manged by AUSTRAC while the other four nodes are managed by four Australian banks - one node each per bank.

Each node is pre-assigned an integer that identifies it. These node identifiers are exposed to the `ftillite` user, so are meaningful. Preferably, these identifiers should be the numbers $0, \ldots, 4$, with 0 being the AUSTRAC node, which we shall henceforth refer to as the *coordinator node*.

The `ftillite` library is composed of two parts: a front end and a back end. The back end (presumably implemented in Golang) executes in each one of the computation nodes, managing and manipulating massive amounts of data, and also communicates this data between the nodes

when necessary. The front end is the library's Python interface. It runs inside a Python interpreter that is located on the coordinator node. The Python objects at the front end initiate commands to the back end that determine how the data is to be manipulated and communicated.

There may also be data communication, in both directions, between the Python front end (with its native Python data) and the back end, but this communication is strictly

- of low volume, and
- only with the back end data that sits on the coordinator node.

This document describes the Python APIs exposed by the library. These are exposed through several object types and several functions.

The Python object types can be divided as follows.

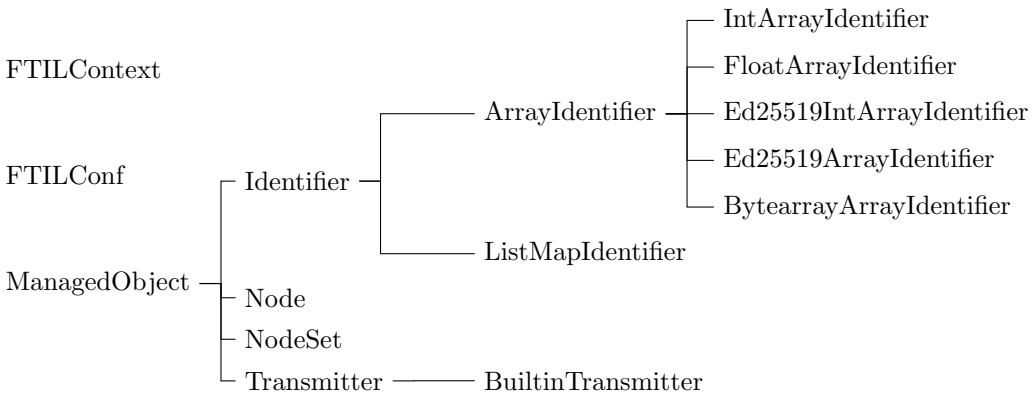
Management objects: This includes a configuration object, `FTILConf`, and a context object, `FTILContext`. These are described in Section 4. All other objects created by the library are created by calls to methods of the `FTILContext` object, and are collectively referred to as *managed objects*.

Data objects: These are proxies to back-end objects that store data. The underlying data can be in one of two data structures, which we refer to as *arrays* and *listmaps*. How these front-end objects connect to back-end data is described in Section 2.6, the structure of the back-end entities is described in Section 2.7, and the front-end Python objects themselves are described in Section 5.

Helper objects: These are managed objects that hold no back-end data. They provide Python interfaces, and may even be implemented as pure Python classes. They include `Node`s, `NodeSet`s and `Transmitter` objects. They are described in Section 2.4, Section 2.5 and Section 2.9, respectively.

Figure 1 shows the hierarchy of types exposed to the user. Though most users will not need most of these type names at all (because they will create new objects by calls to specialised methods rather than by calling their constructors directly), users extending the library will nevertheless need to know the true class names, their constructor signatures, which methods they implement, etc..

Figure 1: The `ftillite` Python class hierarchy



Some notes regarding this hierarchy:

1. We will, for the most part, use the method names that create the objects (which are more visible to the user) as stand-ins for the actual class names. For example, assuming we have an `FTILContext` object `fc`, we will use `fc.listmap` instead of `ListMapIdentifier`.

2. As a special case of the above, we will mainly use `fc.array` despite the fact that there are many different array class types, because a single method creates them all.

3. One reason why we believe the underlying array types should, in practice, be separated, is because the functionality they offer is quite distinct. See Table 2 for an example.

4. This hierarchy represents only those types that are visible to users and which users may want to inherit from. The library may include other types as well, as dictated by software engineering considerations. In this document, for example, we mention the class "`on`", not within the hierarchy. Similarly, concrete `Identifier` types provided by the library also inherit from a class called "`FtilliteBuiltin`". Users should not inherit from the `FtilliteBuiltin` class.

5. The `BuiltinTransmitter` type is used to transmit both `ArrayIdentifier`s and `ListMapIdentifier`s. It should not be inherited by users, however. When introducing new types that can be transmitted, users should inherit from `Transmitter` to create a `Transmitter` type specific for their new `Identifier` type.

In terms of operational structure, the library is designed as follows:

- Work with the library begins by importing the library and creating the context object. This object is responsible for managing the overall system configuration.

- All other objects are created through method calls into the context object, and all are `ManagedObject`s, meaning that they know which context object they relate to.

- Functionality of the library is delivered through three types of interfaces:

  Method calls to managed objects: This is used to manipulate these objects.

  Functions/operators that have managed objects parameters: Whenever at least one parameter required is a managed object, this functionality can be delivered through a global function. We advocate for this when the interface provided by such a function is the most intuitive and/or versatile. The function can still use the context object, by retrieving it from the managed objects.

  Method calls to the context object: This is used for operations that do not have any parameter that is a managed object, such as when generating new managed objects (e.g., empty ones or from randomness).

- Mimicking Python's own use of "magic methods" (which is a solution that allows algorithms to be implemented over multiple, heterogeneous objects of dynamic types) almost all the functionality provided by global functions is merely a very thin Python wrapper around calls to methods of the objects given as parameters. (See Section 6.4 for an example.) Where the methods are never expected to be invoked directly by users, we follow Python's convention of giving them names that begin and end in two underscores. In the case of operators, this wrapper layer is part of Python's built-in functionality.

- Conceptually, some functions/methods/operators expect the parameters sent to them to be compatible to each other in some ways. For example, all `ftillite` functionality that expects multiple managed objects as inputs expects all such inputs to share the same context object. The library has helper functions to facilitate the checking and the coercing of such parameters. Writers of library extensions (as well as programmers of the library itself) can decide to use these convenience functions, or, alternatively, may implement their functionality without automatic coercion (where there are more efficient alternatives). See Section 3.2.

- Every data object (sub-classes of `Identifier`) has an *object scope*, this being a set of computation nodes, stored in a variable of type `NodeSet`. The front-end object manages back-end objects of the same (known) type across all computation nodes within its scope.

- The context object manages the *execution scope*, which is a stack of nested scope definitions. The top of the stack, being the narrowest scope, determines which nodes participate in the current calculation. Unless explicitly noted otherwise, all data objects used as command parameters in a computation must have at least all of the execution scope inside their object scopes.
- By contrast, helper objects have no scope of their own, and can be accessed by any command.

2.2 Initialisation

Working with the library begins by importing it.

```
import ftillite
```

Next, one needs to create a new `FTILContext` object, which manages the connection with remote nodes. The object is instantiated by the command

```
fc = ftillite.FTILContext(conf)
```

where `conf` is a configuration parameter that includes all necessary information, e.g. about the URL and port the object needs to connect to in order to communicate with its back-end (presumably Golang-implemented) services.

Thus, the full initialisation sequence for the library may look something like this:

```
import ftillite as fl

conf = fl.FTILConf().set_app_name("Demo").set_master("https://...:8888")
fc = fl.FTILContext(conf = conf)
```

In order to keep our code examples succinct, we will throughout this document assume that the library has already been initialised, that the FTIL context object is called "`fc`" and that the library itself has been imported under the name "`fl`", as per the example above.

In practice, we expect users to want to import some `ftillite` functionality directly, e.g. by

```
from ftillite import on
```

in order to provide nicer interfaces for themselves than what is given in the examples in this document. However, in this document we will assume nothing has been imported directly, so that all `ftillite` functionality will be highlighted, and so that the reader can easily tell what is a function vs what is a method, and if a method, of which object.

2.3 Managed objects

All objects in `ftillite` other than the `FTILConf` and `FTILContext` are `ManagedObject`s. They are created, directly or indirectly, by the context object. `ManagedObject`s share one method, namely:

`.context()`: This method returns `fc`, the context object related to the `ManagedObject`.

Because extenders of `ftillite` may want to inherit from `ManagedObject`, we also want to specify that its constructor takes only one parameter, this being `fc`, the `FTILConf` object.

For a possible pure-Python implementation of `ManagedObject`, see Section A.1.

2.4 The `Node` class

One type of managed object is the `Node` class. It is a pure Python class that represents a computation node. At minimum, it can be merely a wrapper for a (constant) integer. However, it may be beneficial for it to include other information, such as the name of the node. It provides the following methods:

`.num()`: This method returns an integer that was set for the `Node` object at its construction time.

`.name()`: (optional) If available, this method returns a string that was set for the `Node` object at its construction time.

Other extra methods may also be possible, such as, e.g., a URL specifying where the node is hosted.

It is not possible for users to create new `Node`s directly. However, the user can receive `Node`s from the context object, as described in Section 3.1 and Section 4.2.

2.5 The `NodeSet` class

The `NodeSet` class is a convenience class. It inherits from `ManagedObject` and (at least conceptually) also from the Python native `set` type. It is different from a `set` only in that it validates that all its elements are of type `Node` and all have the same context (which is also the context of the `NodeSet`). `NodeSet`s can be initialised from a single `Node`, from a list or a set of `Node`s, from another `NodeSet`, or, alternatively, an empty NodeSet can be initialised with a parameter that is `fc`, the desired `FTILContext` for the object. If a `NodeSet` is initialised from a single `Node`, this is interpreted as instantiating a single-node set containing only this node.

For a possible pure-Python implementation of `NodeSet`, see Section A.2.

2.6 `Identifiers`

`Identifier`s are front-end Python objects that manage back-end data-holding objects. The front-end objects are not distributed. They follow standard Python semantics. However, each `ftillite Identifier` manages a *handle* that uniquely identifies a back-end object and defines a *scope* which is the set of nodes (`NodeSet`) in which the back-end object exists. This can be any non-empty subset of the nodes in the system. Back-end objects exist with independent values in each of the nodes in their scope, but all share the same type, which is determined by the type of `Identifier`. (The `Identifier` class itself is never directly instantiated.)

Note 2.1. Because the abstract `Identifier` object actually has no idea what type of back-end object it manages, the code of the abstract class should not include explicit handling of a "handle". See Section 8 for examples of user-defined types deriving from `Identifier`, and how they do not and cannot utilise any handle-management code that may exist in `Identifier`.

An `Identifier`'s type, handle and scope are set at its construction time and do not change later. (The handle may change in ways that are transparent to the user. See Section 4.3.2 for an example.)[1]

The identity of the handle and the back-end object it identifies are never exposed to the Python layer, so can be implemented in any way. For example, handles may or may not be integers. The same is true for the data payload carried by each back-end object within its definition scope.

The `Identifier`'s scope, however, is exposed by the following method, shared by all `ftillite Identifiers`:

`.scope()`: This method returns the `Identifier`'s definition scope, as a `NodeSet` object.

[1] For `Identifiers` of type `BytearrayArrayIdentifier`, the `.size()` attribute of the stored type is also set at construction time and cannot be changed later on.

Extenders of `ftillite` will also be able to inherit from `Identifier`, for which reason we will specify that its constructor expects only `fc`, an `FTILContext` object.

FTIL Note 2.1. The back-end storage, which may be implemented in Go, is akin to FTIL "segment managers". The front end is APIs that create, manipulate and delete the back-end objects. These APIs are executed by Python commands, via Python's standard variable management and flow control, which acts here as a "compute manager".

All `ftillite` objects behave like FTIL "distributed" types. There is no distinction between "distributed" and "shared" types in `ftillite`, because that distinction is purely about security guarantees provided by FTIL.

The handles in `ftillite` are significantly more simplified than those of FTIL, which causes both unnecessary data movement and some privacy issues. For an example of privacy issues, see Section A.5.

The two main types of `Identifier`s are `ArrayIdentifier`s and `ListMapIdentifier`s. They are described in Section 5.

Essentially, `ArrayIdentifier`s have a remote payload that is a list of elements. The elements are all of the same type in every node, but there may be a different amount of them, including zero, in each node.

The `ListMapIdentifier`'s payload is a more specialised type. It is used as a building block in implementing distributed maps and distributed sets. It, too, may be empty in some or all of the nodes in its scope.

If the payload of an `ArrayIdentifier` is a single value in each node in scope, we call this a *singleton*. Singletons are often used in `ftillite` to represent scalar arguments in operations.

All `Identifier`s support the "`.typecode()`" method. For `ArrayIdentifier`s, this is a method that provides, in string form, the type of elements managed by the `ArrayIdentifier`'s lists.

The following types are supported (given here with their "`.typecode()`" results):

`typecode = 'i'`: `int`. A 64-bit signed integer.

`typecode = 'f'`: `float`. A "double"-sized floating point variable.

`typecode = 'I'`: An `Ed25519Int`.

`typecode = 'E'`: An `Ed25519` group member.

`typecode = 'b`*k*`'`: `bytearray`: A byte array of length *k* bytes, where *k* is given as an explicit literal integer. This type is used for, e.g., symmetric encryption and hashing results.

As can be seen, mostly this typecode reflects the sub-type of `ArrayIdentifier` used, but in the case of `BytearrayArrayIdentifier` it is even more specific than this, including the specific size of `bytearray` being managed.

We refer to the first letter of the typecode as the *base typecode*. In the `ftillite` type hierarchy, each base typecode corresponds to a specific `ArrayIdentifier` sub-class.

More generally, an `Identifier`'s "`.typecode()`" is a hint regarding how the type behaves when serialised/de-serialised. All `Identifier`s can be serialised and communicated as a list of standard `ftillite` `ArrayIdentifier`s (usually of equal length, as would be required to support reading from a database, regarding which see Section 2.8). In the general case, the typecode of an `Identifier` is a string composed of the typecodes of every standard `ftillite` `ArrayIdentifier` type it creates when serialised. (Regarding this type of serialisation, see the `Identifier` methods "`.flatten()`" and "`.unflatten()`". These are unrelated to the methods "`.serialise()`" and "`.deserialise()`", which provide serialisation into a `BytearrayArrayIdentifier`.)

2.7 Back-end object implementation

When a new front-end `Identifier` object is created, such as via calls to

```
my_array = fc.array(... some initialisation here...)
```

or

```
my_listmap = fc.listmap(... some initialisation here...)
```

this initialisation creates a back-end object of the appropriate type, scope and content payload, as well as the front-end Python object that serves as a proxy to it. The front-end Python object (which, like all Python objects, can be referred to by any number of Python variables) is a local, non-distributed data type, which really only houses a handle identifying the back-end object. When `ftillite` functions are called, they are sent Python objects as parameters, and `ftillite` interprets this as instructions to manipulate the underlying, distributed back-end objects that are identified by those handles.

When an `ftillite` object's destructor is called (which is something that happens to a Python object whenever that object is no longer referred to by any variable), this triggers erasure of the back-end object that the handle refers to, throughout that object's entire scope.

FTIL Note 2.2. Albeit somewhat simplified, this structure mirrors FTIL's referencing structure, with FTIL "entities" being back-end objects, FTIL "variables" being handles, and FTIL "identifiers" being Python variables.

FTIL's semantics go to extreme lengths to avoid unnecessary copying. In `ftillite`, we've opted for simplified semantics, at the cost of more data movement (and some increased privacy risks, as demonstrated in Section A.5).

Early releases of FTIL included "local" objects, in addition to other flavours like "shared" and "distributed". In `ftillite`, the role of "local" objects is taken up by native Python objects.

2.8 `AuxDB`

Each computation node in the system is associated with a database, which we will refer to as `AuxDB`. Some `ftillite` commands will be used to invoke SQL commands, in parallel, on the databases associated with each node in the execution scope, and will use the query results to populate the payloads of distributed back-end objects. Each node's portion of the back-end object will be populated only by the results from that node's own SQL database.

FTIL Note 2.3. The `AuxDB` is `ftillite`'s version not only of FTIL's Auxiliary DB but also of its Transaction Store and user input area. For the purpose of the PoC, we assume that all data is known in advance, so no user input is needed. Furthermore, because FTIL's security guarantees are not enforced, there is no need to separate the Transaction Store from the Auxiliary DB or to provide any access protection to the `AuxDB`.

2.9 The `Transmitter` type

All inter-node communication of back-end object data is mediated in `ftillite` by `Transmitter` objects.

The `Transmitter` type is a helper type. It has an underlying transmitted object type that is given to it at construction time (typically, the type of the object that created it using a call to its method "`.transmitter()`"). This underlying object type cannot be externalised. The `Transmitter` only supports one method.

`.transmit(dictionary)`: The parameter `dictionary` is a Python dictionary. Its keys are `Node`s, being node identifiers. Its values are all of the `Transmitter`'s underlying type. The execution scope for `transmit` must include at least all the dictionary keys and all the scopes of the dictionary values. The return value is another Python dictionary of the same type, such that any value that was in `dictionary[n1]` on node `n2` is in the return value on node `n1` in the value that is under the key `n2`. Keys that do not carry values in any node are omitted from the returned dictionary altogether. In terms of implementation, any value that is initially

in node `n1` and ultimately in node `n2` must only be revealed to these two nodes. In terms of security requirements, communication between these nodes must be "direct", in the sense that the the sender can be sure of the identity of the receiver, the receiver can be sure of the identity of the sender, and both can be sure that the message was not corrupted in transit.

Note 2.2. The `Transmitter` object is not an instance of the `Identifier` type. It has no "`scope`". For this reason, regardless of what the execution scope was at the time of the `Transmitter`'s creation, it can be used in any scope.

The dictionary values that are the parameters to `.transmit()` may have a narrower scope than the current execution scope, this being a major exception to `ftillite`'s standard parameter processing.

FTIL Note 2.4. Even though in recent versions of `ftillite` transmitting is now managed by a `Transmitter` object, it is still the case that the way `transmit` is handled remains one of the greatest points of divergence between FTIL and `ftillite`. In FTIL, items in `dictionary[n1]` on node `n1` are never copied. In `ftillite` they always are. But for this FTIL has safety mechanisms that are absent from Python. We opted for the safer and simpler implementation in this case, at the expense of some reduced efficiency in such edge-cases. (Mostly, users can avoid having such values in the dictionary in the first place.)

There is no functional requirement for `Transmitter`s to have any specific class hierarchy. The only requirement is that each `Transmitter` can only transmit objects of its designated underlying type, meaning objects of the exact Python type and the exact typecode specified. For example, all built-in `Identifier` types are transmitted using a single `BuiltinTransmitter` type. However, each instance of `BuiltinTransmitter` can only `transmit` elements of one specific type and typecode. We recommend for extension types created by users, however, to create a separate `Transmitter` sub-type for each transmittable object type.

The existing class hierarchy lends itself to the following straightforward implementation of the "`.transmitter()`" method for built-in types:

```
def transmitter(self):
  return BuiltinTransmitter(self)
```

assuming an appropriate constructor for the `BuiltinTransmitter` type.

The specific choice of class hierarchy under `Transmitter` is not a mandatory part of the API, however. An alternative implementation would have created separate `ArrayTransmitter` and `ListMapTransmitter` subtypes, or even had a separate `Transmitter` subtype for every concrete `ArrayIdentifier` subtype.

Note 2.3. There is fairly little reason to actually have a common parent, `Transmitter` for all `Transmitter`-type objects. In our reference implementations (as well as in our examples when extending the library) all `Transmitter`s derive from the `Transmitter` class, but this class provides no (non-abstract) functionality. We could have equally programmed these to inherit directly from `ManagedObject`, rather than from an intermediate type.

3 Command execution

3.1 The execution scope

Commands in `ftillite` execute in a particular *execution scope*. This is the set of nodes that are actually impacted by the commands executed. The rest of the nodes are unaffected.

Unless otherwise specified, every command requires all of its operands to have a scope that is a superset of the current execution scope, or this results in a `KeyError` exception being raised.

The current execution scope can be retrieved by the command

```
fc.scope()
```

Its return value is a `NodeSet` object. Its default value is the set of all nodes.

The execution scope is the top element in an *execution scope stack*.

The stack can be manipulated by creating a new context, as follows:

```
with fl.on(new_scope):
  # The block inside this "with" executes in "new_scope". This value is added
  # to the top of the stack at the start of the block, and removed at its end.
```

FTIL Note 3.1. This handling of scope mirrors FTIL's stack-based execution, with FTIL's "`on(scope):`" directive directly replaced by the Python syntax "`with fl.on(scope):`".

Also available is the following syntax:

```
with fl.on(new_scope) as scope:
  # This is the same as above, except that the variable "scope" now holds the
  # new value of fc.scope().
```

The new scope cannot be empty and cannot include any nodes that are not within the present scope. The scope can be provided as a `NodeSet`, a set of `Node`s, as a list of `Node`s, or as a single `Node`. The input is converted to a `NodeSet`.

For a pure-Python implementation of the "`on`" directive, see Section A.3.

3.2 Automatic parameter manipulation

When running any `ftillite` command, some manipulation and some checking is always done to its parameters. The following section describes these operations, in the order in which they occur. In some cases, as indicated, the library will provide tools that facilitate performing these operations, in order to help users wishing to extend the library.

3.2.1 Context checking

All global `ftillite` functions except the constructors of `FTILConf` and `FTILContext` expect at least one `ManagedObject` among their parameters. It is first tested that all such `ManagedObject`s share the same context object. Usually, this context object's identity must also be retrieved explicitly, as it is needed for the implementation of the function.

When running methods of `ManagedObject`s it is similarly verified that all `ManagedObject`s given as parameters have the same context object as the method-running object.

Where this is not the case, an exception is raised.

To facilitate extension of the library, all this can be implemented by

```
fc = fl.get_context(params)  # Retrieve the context from a ManagedObject among
                             #   the arguments in "params".
fc.verify_context(params) # Verify that all ManagedObjects in "params" have the
                          #   same context.
```

Multi-parameter functions will almost always run one of the two commands above, but generally not both: "`get_context`" also verifies the context before returning `fc`, rendering any subsequent invocation of "`verify_context`" superfluous.

We will assume, throughout the rest of this section, that when invoking a global `ftillite` function, its arguments are always processed in the manner described above, and so `fc` is known and has been verified to be common to all `ManagedObject` arguments.

Where a method is invoked, we assume that the following alternate process was invoked, with the same effect:

```
self.context().verify_context(params)
```

3.2.2 Scope checking

With the exception of the "`fl.transmit`" function, all global `ftillite` functions and all `Identifier` methods require the calling object itself and any `Identifier` parameters to have a definition scope that is a superset of the current execution scope.

This is tested and a `ValueError` exception is raised if it does not hold.

Note that the current execution scope can be determined by running "`fc.scope()`", and the identity of `fc` was determined in the context checking stage (Section 3.2.1).

3.2.3 Value promotion

Some functions, operators and methods in `ftillite` take multiple `ArrayIdentifier` parameters, and of these parameters there are some that need to be arrays of the same typecodes for the operation to be meaningful. This is the case, for example, when using the `array __add__` operator.

Given a set of parameters that all need to be of the same type and typecode (which usually also determines the result type), `ftillite` employs the following strategy. Note that the parameters actually given may not be `ArrayIdentifier`s at all.

1. As per Section 3.2.1, it is verified that at least one of the arguments is a `ManagedObject`. The identity of `fc` is extracted from this argument and it is verified that all other `ManagedObject`s in the input share the same context.
2. The current execution scope, "`fc.scope()`" will be the scope for any `ManagedObject` generated in this step.
3. Any Python integer is interpreted as a singleton `IntArrayIdentifier`.
4. Any Python float is interpreted as a singleton `FloatArrayIdentifier`.
5. If the typecode of all inputs is the same, no further promotion is required, and promotion processing stops here.
6. Otherwise, if the inputs include both integer and float `array`s, all `IntArrayIdentifier`s are converted to `FloatArrayIdentifier`s.
7. Otherwise, if the inputs include both integer and `Ed25519Int array`s, all `IntArrayIdentifier`s are converted to `Ed25519IntArrayIdentifier`s.
8. Otherwise, a `TypeError` exception is raised.

Wherever a parameter's initial type is also its final type, no conversion and no copying takes place. Rather, the original value is used.

Note 3.1. The instructions above may be exactly what is implemented for the proof of concept, but longer term it should be noted that only the final result of operations is important in this specification. Mostly, it is possible to implement the `ftillite` library in such a way that operations requiring promotion do not need to actually materialise for this new `fc.array`s. However, the system must still behave as though an actual conversion has taken place. For example, if an integer value cannot be converted into a float without loss, this should raise an exception. (See Section 5.2.3.)

A somewhat different process is used in operations that involve an assignment (e.g., "`+=`"). In such operations, the left-hand-side parameter must be an `ArrayIdentifier`, and no promotion occurs on its type. Instead, the other parameters are promoted to this result type (i.e., converted from int to float or from int to `Ed25519Int`, where applicable). If this is not possible to do, a `TypeError` exception is raised.

Instances of (the sub-classes of) the `ArrayIdentifier` type provide the convenience method "`.promote(typecode)`", to allow library extenders to mimic this functionality, and the `FTILContext` object provides a wrapper for this, too. (An exact specification of this wrapper is given in Section 4.3.6.)

Note 3.2. Python provides a helper function of its own for value promotion, namely the magic method "`.__coerce__()`". Whether `ftillite` promotion makes use of Python's coercion mechanism or not is an implementation choice, and beyond the scope of this document.

3.2.4 Automatic value broadcasting

The `ftillite` library supports automatic value broadcasting, similar to `numpy`'s. This works as follows.

Some functions, operators and methods in `ftillite` take multiple `ArrayIdentifier` parameters, and of these parameters there are some that need to be arrays of the same length for the operation to be meaningful. This is the case, for example, when using the `listmap __getitem__` operator.

To describe the procedure that `ftillite` uses in order to handle such cases, let us first describe the method "`fc.calc_broadcast_length(params)`".

This method of the context object accepts in "`params`" a Python list of objects. It performs the following.

1. First, the method determines the lengths of all elements in `params`. An element of a native float or int type is considered to have length 1 in every node in scope. Otherwise, the length is determined by invoking the method "`.len()`" on the parameter. The return value is checked to be a singleton `IntArrayIdentifier`. (Note that elements in `params` may be of user-defined types, which is why this check on the return value is important.)

2. If all lengths are "1", the return value of the function is a singleton `IntArrayIdentifier` whose value is 1.

3. Otherwise, the method calculates in a singleton `IntArrayIdentifier` the maximum of all the lengths of all parameters whose length is not 1. This value is then returned as a singleton `IntArrayIdentifier` value.

4. Note that the above computation is performed separately on each node. For example, one node may have all lengths equal 1 and the return value in that node will be 1, while in another node there will be "0"s and "1"s and the return value will be 0. At the same time, in yet a third node the maximum will be 2, and there the return value will also be 2.

When an `ftillite` function is sent parameters that need to all be of the same length, the exact process in which the function handles this is a matter of implementation (and hopefully in some cases it is more optimised than what is described below, triggering less data copying). Semantically, however, what is performed is the following.

Let `params` be the list of arguments to the function that all must be of the same length.

1. The context, `fc`, is retrieved as described in Section 3.2.1.

2. The function executes "`length = fc.calc_broadcast_length(params)`".

3. For each element in `params`, if the element, `x`, is of type int, it is sent to the function as `(fc.array("i", length, x), True)`.

   If it is of type float, it is sent as `(fc.array("f", length, x), True)`.

4. Otherwise, what is sent to the function is "`x.broadcast_value(length)`".

The Boolean appearing as the second tuple item provides the downstream function with the information of whether the handle received is a copy or not.

Note 3.3. This process allows `x` to be of a user-defined type, as long as that type implements the methods "`len`" and "`broadcast_value`".

The `calc_broadcast_length` functionality is not a convenience function. It is important that it is provided by the library, because attempting to implement it by running `ftillite` functions directly would trigger recursive value broadcasting.

A somewhat different process is used in operations that involve an assignment (e.g., "`+=`"). In such operations, the left-hand-side parameter must be an `fc.array`, and no broadcasting occurs on it. Instead, the other parameters are broadcast to its length. If this is not possible to do, the result is a `ValueError` exception.

3.3 Exception handling

Note 3.4. The material in this section is best delved into after one already has familiarity with the rest of the `ftillite` API.

FTIL Note 3.2. This section has no direct parallel in FTIL, because in FTIL exceptions cannot be caught, making their handling much more straightforward.

Some conditions may trigger an `ftillite` exception. This can happen in the Python layer before ever forwarding commands to the back end (e.g., when a command tries to manipulate a variable that is not in scope), in the back end (e.g., if the system runs out of storage space when trying to execute a command) or in any other step in the process.

Regardless of where the error occurs, it is translated into a Python exception with all necessary information (e.g., which node reported an error), which the user can then catch and handle as usual.

We distinguish between two types of exception situations:

User errors are exceptions that are the result of a usage error on the part the `ftillite` user: a command was invoked in a situation where that command was not allowed or its parameters are invalid, and

System errors — all else. Typically, these will include errors that are the result of the practicalities of the system environment: network errors, GPU not found, cannot allocate memory, hard-disk full, no write permission to database, etc..

The rule regarding all errors is that they should not leave the system in a non-coherent state. For example, `listmap`s should not suddenly have values that are not consecutive integers.

If an exception occcurs, operations that normally generate new handles do not create these handles. Ideally, any variables stored with existing handles should not change their values due to an operation that failed with an exception.

For general errors, and, in particular, for system errors, this is impossible to guarantee without full transaction support, but we do not require such support (and, indeed, do not want to pay the performance penalties of one). For this reason, we separate the handling based on the operation type and based on the exception type, as follows.

1. If an operation does not normally modify a given handle, that handle must not be modified even if an exception occurred.

2. Otherwise, if a system error occurred, any handles that would have been modified by the operation if successful must at least remain in a coherent state. It is OK for `array`s to become empty arrays and for `listmap`s to become empty `listmap`s. It is also OK for these handles to be deleted completely. However, if the handles are not deleted, the underlying variables must not change their type and typecode.

3. Ideally, handles should retain their previous values if possible. If this is not possible, the next best alternative is for `array`s to retain their original size if the operation that triggered the exception is one that does not normally change the `array`'s size. Otherwise, retention of the original size is not advantageous, unless also retaining the original values.

4. In the case of user errors, if a `listmap` was to be modified, that `listmap` *must* retain its original value. (This mainly impacts the implementations of the methods "`.add_items()`" and "`.remove_items()`", for which it is necessary to first perform a full prerequisite check on the inputs, prior to modifying the target `listmap`.) If an `array` was to be modified, it is preferable that this `array` retains its original value, but that is not mandated. Mandated is only that if the original operation was one that does not normally change the `array`'s length, then the array should retain its original length.

Note that in some cases, and in particular in order to handle `listmap` "`.add_items()`" and "`.remove_items()`", it is necessary for the compute manage to initiate two commands in response to a single user `ftillite` command: first, all prerequisites must be checked on all nodes in scope, and only if all nodes in scope agree that the operation is legal can it be executed. If this is not done, it is possible that one node will trigger an exception and will return the original `listmap` data, whereas others will not detect the error and will modify the `listmap`, contrary to the desired result as specified in this section.

4 The context object

The `ftillite` library has two management objects: the `FTILConf` configuration object, and the `FTILContext` context object. The entire purpose of the configuration object is to aid in constructing the context object. This, we have already discussed in Section 2.2, so we will not flesh out `FTILConf` further here. The rest of this section defines the `FTILContext` object.

FTIL Note 4.1. This parallels the FTIL context object described in [1], Section 2.12.

4.1 Constructors

The `FTILContext` objects has only one constructor. It was demonstrated in Section 2.2:

```
fl.FTILContext(conf)
```

It expects an `FTILConf` configuration object.

4.2 Attributes

The context object, `fc`, will have the following attributes. which act as "constants" (even though Python does not enforce constness).

```
fc.CoordinatorID  # The identity of the coordinator node, of type Node.

fc.MyID # Of type IntArrayIdentifier and having all nodes in its scope: a
        #   length 1 array in each node that provides the integer node
        #   identifier of the node.

fc.Nodes # A convenience variable, equivalent to
        #   {n.num() : n for n in self.MyID.scope()}
```

FTIL Note 4.2. This parallels the FTIL constants.

4.3 Methods

The `FTILContext` object provides the following methods. We divide their description according to their type.

4.3.1 General data object creation

`fc.array(...)`: Creation of a new `ArrayIdentifier`. Described in Section 5.2.1.

`fc.listmap(...)`: Creation of a new `ListMapIdentifier`. Described in Section 5.3.1.

As a general rule: unless otherwise stated, both here and everywhere else, any creation of a new `Identifier` is always done such that the new object's definition scope is the current execution scope.[2]

4.3.2 Session storing and restoring

In order to allow one to close the Python session and then to reconnect at a later time, the context object provides means to store both the local information (the Python object structure) and the remote information stored by the data objects.

To store, the Python user specifies one Python object to store, where the expectation is that all objects that require storing are linked to this one Python object.

FTIL Note 4.3. This mirrors the recommendation in [1] Section 11.2.1 to keep all objects belonging to one investigation under a single object.

`fc.save(obj, destination)`: Here `obj` is the Python object to be stored and `destination` is a string specifying under what name to store it. There is no expectation for any part of this saving to be done outside the confines of the system, so the name is internal and doesn't need to correspond to any URL, database table, or any other concrete "location". The command performs a Python "pickle" to store the object with all its attributes, recursively. `Identifier`s trigger by this a saving of all their remote data, in a way that retains the information of both the save destination and the specific handle associated with each `Identifier`. Note that the remote data cannot simply remain in place, as it is still connected to `Identifier`s. (Such `Identifier`s may change the data's contents after a `save` operation, which would clearly be undesirable.)

`fc.load(destination)`: Reverses the `save` operation. The returned value is an object semantically identical to the original `obj`, with the exception that all `ManagedObject`s get the `FTILContext` object that loaded them as their new context.

`fc.delete(destination)`: Deletes the pickle stored in `destination`, as well as all remote data associated with it.

4.3.3 DB i/o

FTIL Note 4.4. This part of the `ftillite` specifications follows the object-creation functionality of Section 3.11 of [1].

`fc.auxdb_read(query, typecodes)`: `query`: A SQL query, given as a string. May not have any return value, or may return a table with any number of columns.

`typecodes`: A Python string concatenating as many typecodes (with any associated size specifications) as there are columns in the return table of the SQL query. Whitespaces are allowed before, after, and between typecodes, but not within typecodes.

The command runs, separately in each node in the current execution scope, the SQL query on its local `AuxDB`. If `typecodes` includes only a single typecode, the result is returned in the form of an `fc.array` of this typecode and scope. If there are multiple typecodes in `typecodes`, the result is returned as a Python list of `fc.array`s, whose individual typecodes are as specified by `typecodes` and whose scope is the current execution scope. The contents of each returned `fc.array` represent the values of a single column of the table returned,

[2] The one exception is when `transmit`ting objects.

converted to the appropriate type. If the query does not return a result, the Python function does not return a value. This can be used to run arbitrary SQL code in the database. If there is a discrepancy between the number of columns returned by the SQL query and the number of typecodes specified in `typecodes`, or if the query results cannot be parsed according to `typecodes`, this raises a run-time exception.

`fc.auxdb_write(table_name, col_names, data): table_name:` A Python string, signifying the name of an existing table in the `AuxDB`.

`col_names:` A Python list of Python strings, signifying the names of columns in the table specified by `table_name`.

`data:` A Python list of `ArrayIdentifier`s. The Python list should have the same length as `col_names`, and the `array`s in the list should all have the same length (after any broadcasting). Furthermore, the `array` types should match the types of the columns in the specified table.

The command executes an "`INSERT INTO`" SQL command that takes the data from the specified `array`s and pushes it into the specified columns of the specified table. This command is important in that it is the only command that allows local `ftillite` variable data to be written back into the local `AuxDB`. In this command, each node in scope works independently, writing only the variable data local to it. It is an error to run this command on any `array`s whose scope is narrower than the scope of the command. Note that the expectation is that the creation of the table and the setting of any table properties will be done using calls to `auxdb_read`. It is only the "`INSERT INTO`" that is done using `auxdb_write`.

4.3.4 Random values

The context object has multiple means for generating random values. Many of these are supported using the method `fc.randomarray`. The general signature of this function is as follows.

`fc.randomarray(typecode, length,` *optional additional parameters*`)`

In general, `typecode` is a typecode string, `length` is a singleton integer `fc.array` (or a Python integer, which then gets automatically promoted to one), and the result is, in each node in the execution scope, an `fc.array` of the specified type and length, such that each of its elements is an independent random value allotted from the distribution defined by the additional parameters. Elements are independent both from each other within a single `fc.array` and between elements on different nodes.

Here is how the target distribution is specified, split according to the base typecode.

`'i':` The parameters are "`minimum`" and "`maximum`", which are singleton integer `fc.array`s (or Python integers, which are then promoted). The distribution implied is the uniform distribution in `[minimum, maximum]`.

`'f':` As above, but `minimum` and `maximum` are singleton float `fc.array`s (or automatically promoted Python floats).

`'I':` The parameter is "`nonzero`", a Python Boolean, with a default value of `True`. If `False`, the distribution is uniform among all `Ed25519Int` values. If `True`, the distributed is uniform among all nonzero values.

`'b':` No parameters. The distribution is uniform among all possible values of the given type.

Note 4.1. The process by which random floating point numbers are to be generated is

```
return minimum + ((maximum - minimum) * float64(RandomInt63())) / (1 << 63)
```

Arguably, this specific generation process creates some avoidable artefacts, but users wanting cleaner random floating point numbers will need to implement the generation of such numbers themselves.

Additionally, the context object has the following method:

`fc.randomperm(length, n = length)`: The parameters `length` and `n` are both either integers or singleton integer `fc.array`s. If `n` is specified explicitly, its value in each node within the execution scope must be no smaller than the value of `length`. The function returns an integer `fc.array` of the specified `length`, whose values are uniform in `[0, n - 1]` and independent (both between each other and between values in other nodes), subject to the restriction that no two elements in any single node can be the same.[3]

4.3.5 Cryptographic functions

Note 4.2. As described in other FTIL documents, the requirement is really for secure cryptographic primitives of various types, not for specific algorithms. We suggest here particular algorithms for concreteness and in order to demonstrate the full syntax.

The `FTILContext` is used to generate all types of keys. For symmetric keys it provides the following methods:

`fc.aes256_keygen()`: Generates an AES key.

`fc.grain128aeadv2_keygen()`: Generates a Grain128 key.

The above may all be just wrappers over `randomarray`. However, by separating them into their own methods we allow the possibility of checking for and eliminating any potential weak keys in each algorithm.

Note 4.3. We have not included special methods for generating initialisation vectors (IVs) for cryptographic algorithms. The assumption is that these can be generated using `randomarray`. If this assumption proves incorrect, specialised methods will be required for these, too.

Symmetric keys are simply stored as singleton `BytearrayArrayIdentifier`s of the appropriate size.

For non-symmetric keys, `fc` provides the following.

`fc.ecdsa256_keygen()`: Generates a private/public key pair for the ECDSA digital signature algorithm. (Other 128-bit secure digital signatures can be implemented instead.) One pair is generated for each account in the command's execution scope. The result is returned as a tuple of `bytearray` singletons.

`fc.rsa3072_keygen()`: Generates a private/public key pair for the 3072-bit RSA algorithm. (Other 128-bit secure public key algorithms can be implemented instead.) One pair is generated for each account in the command's execution scope. The result is returned as a tuple of `bytearray` singletons.

Note that all key generating methods can only generate one key at a time.

4.3.6 Miscellaneous methods

`fc.scope()`: Returns the current execution scope as a `NodeSet` object.

`fc.calc_broadcast_length(params)`: See Section 3.2.4.

[3] Python does not actually allow the default value of `n` to be `length`. Instead, the default value is set technically to be `None`, and `n` is simply interpreted to be `length` if it is not explicitly specified.

`fc.verify_context(params):` Given a Python list, go over each element in the list. If the element is a `ManagedObject`, verify that its "`.context()`" is `fc` itself. (It is allowed for none of the elements to be `ManagedObject`s.)

`fc.promote(obj, typecode = None):` Convenience method, equivalent to:

```
is_copy = False
if type(obj) is int:
  obj = self.array("i", 1, obj)
  is_copy = True
elif type(obj) is float:
  obj = self.array("f", 1, obj)
  is_copy = True
if typecode != None:
  (obj, is_copy2) = obj.promote(typecode)
  is_copy = is_copy or is_copy2
return (obj, is_copy)
```

`fc.arange(length):` Creates an integer array with the numbers 0 through *length* − 1. The "`length`" parameter may be an integer or a singleton `fc.array` of integers.

5 Data managing objects

This section returns to the `Identifier` class, already introduced in Section 2.6. This is the top-level Python class whose members effectively manage any distributed data in the system.

It has two immediate derived classes, `ArrayIdentifier` and `ListMapIdentifier`.

FTIL Note 5.1. While the `ArrayIdentifier` type serves to replace FTIL's fixed-sized types, as well as its `list` and `memblock` container types, the `ListMapIdentifier` type allows a user of this API to regain the functionality normally gained from FTIL's `dict`, `set` and `hashmap` types.

The rest of the data in the system (e.g., strings and references) is entirely on the coordinator node, and is managed by the Python front-end.

5.1 The `Identifier` type

Having a common ancestor to `ArrayIdentifier` and `ListMapIdentifier`, as described in the `ftillite` class hierarchy in Figure 1, is transparent to the everyday user of `ftillite`, but visible to anyone extending the library who may wish to inherit from it. For an example, see Section 8.

The following are methods that all concrete `Identifiers` support. We delineate specifically those that are abstract methods. In practice, other than "`.context()`" (which is inherited from `ManagedObject`) and "`.scope()`" (which is the core new property of an `Identifier`), any functionality that exists in an `Identifier` and is not abstract, should be taken to be a "default implementation". Inheriting types will very likely want to replace these default implementations by more efficient implementations, tailored to their specific type's needs.

A possible pure-Python implementation of `Identifier` appears in Section A.4.

`.context():` Returns the context object `fc`. (This method is inherited from `ManagedObject`.)

`.scope():` Returns the scope in which the `Identifier` is defined. This is a `NodeSet` object.

`.transmitter():` Though this method has no implementation for an abstract `Identifier`, every concrete subclass of `Identifier` implements a `.transmitter()` method that returns a `Transmitter` object suitable for communicating values of the calling object's type. See Section 2.9 for details.

`.stub()`: Abstract method: return a new object of the same type and typecode, but empty. For an `ArrayIdentifier` this is simply "`return self.context().array(self.typecode())`" and for a `ListMapIdentifier` this is "`return self.context().listmap(self.typecode())`".

`.sametype(other)`: Abstract method. Returns a Python Boolean to indicate whether `self` (the left-hand side) and `other` are of the same type (meaning, e.g., that they can be exchanged in the same `transmit` command).

`.flatten()`: This method is abstract here, to be implemented in derived classes. Its purpose is to represent the information in an `Identifier` as a Python list of `ArrayIdentifier` elements, e.g. for later use as a `ListMapIdentifier` key.

`.unflatten(data)`: This method is abstract here. The parameter `data` is a Python list of `ArrayIdentifier` elements. The method is meant to be an inverse operation to `.flatten()`: the information from `data` overrides the original `Identifier`'s information and replaces it by a state such that "`self.unflatten(data).flatten()`" is always equal to `data`. This is done without changing the nature of the `Identifier`. For example, the method does not alter the `Identifier`'s typecode. The return value from `unflatten` is the new object itself.

`.width()`: Has a default implementation here, but derived classes can provide their own, more efficient implementations. Is semantically equivalent to "`return len(self.flatten())`", but see Section A.4 for an alternative default implementation.

`.auxdb_read(query)`: This method runs the SQL query in the Python string `query` on AuxDB, reinterprets the result as an `Identifier` of the appropriate kind, and stores it in `self`, overwriting any previous value. See sub-types for details and Section A.4 for a default implementation.

`.typecode()`: Returns the types of columns produced by "`.flatten()`". Has a default implementation here as "`return "".join([x.typecode() for x in self.flatten()])`", but derived types are free to replace this by a more efficient implementation. Specifically, this method *must* be overwritten by all the standard `ftillite` concrete `ArrayIdentifier` subtypes.

`.copy()`: Abstract here. This method returns a deep copy of the original `Identifier`. The result has a new handle and is defined on the nodes in the current execution scope (which, as usual, must be a subset of `self.scope()`).

`.__setitem__(key, data)`: Abstract here. Not all `Identifier`s support item setting, but all should support a key value that is `slice(None, None, None)`. In this case, the new data overrides all existing data in the `Identifier`. Note that this operation is done in-place: the handle does not change, nor can the `Identifier`'s type and typecode change. (For types that do support item setting, the behaviour of using `key = slice(None, None, None)` may be quite different to the behaviour with other `key` values. An example of this is `ArrayIdentifier`s, where this special value of `key` is the only value that can change the length of the `ArrayIdentifier`.)

FTIL Note 5.2. The implementation of "`__setitem__`" as above is enough to support the syntax

```
a[:] = b
```

This syntax serves in `ftillite` the same role as sub-assignment in FTIL.

5.2 The `ArrayIdentifier` type

`ArrayIdentifier` is a Python type that encapsulates a list of a homogeneous data type. Each `ArrayIdentifier` has a `typecode` string that uniquely identifiers the list's scalar type, as follows:

`typecode = 'i'`: `int`. A 64-bit signed integer.

`typecode = 'f'`: `float`. A "double"-sized floating point variable.

`typecode = 'I'`: An `Ed25519Int`.

`typecode = 'E'`: An `Ed25519` group member.

`typecode = 'b`k`'`: `bytearray`: A byte array of length k bytes, where k, the *size* specification, is given as an explicit numeral literal. This type is used for, e.g., symmetric encryption and hashing results.

The "size" for `bytearray` arrays is a single, positive integer for the entire array, determined at initialisation time. It determines how many bytes are in each array element.

For example, the typecode for an array of 32 bytes is `'b32'`. This is how it is set and how it is retrieved when queried. We will refer to the `'b'` in this case as the *base typecode*.

In the Python class hierarchy given in Figure 1, each base typecode has its own class, derived from `ArrayIdentifier`.

The elements of an array are numbered sequentially from zero.

FTIL Note 5.3. There is no native `ftillite` support for FTIL's composite fixed-sized types (like pairs). Instead, a user is meant to create multiple arrays of atomic fixed-sized types. Atomic fixed-sized types that are not represented, like Booleans, are represented through integers. (In the case of Booleans, this is done in the usual way, with non-zero indicating "True".)

The `bytearray` type existed in early versions of the FinTracer Compendium. It is used as a generic type that can store, e.g., hash results and block cipher results, and on which bitwise-Boolean operations can be performed.

Note 5.1. Some of the arrays used in `ftillite` will have very large lengths. There is no requirement for arrays to be contiguous in memory or on disk. Chunking them beyond a certain size is probably advisable.

See Section A.5 for a Python implementation of `ArrayIdentifier`.

5.2.1 Constructors

The `ArrayIdentifier` class itself is an abstract class. As such, users never run its constructor directly. We list the constructor here because it is useful in the context of inheriting from `ArrayIdentifier` in order to extend the library. Examples can be found in Section 8.

The signature for the `ArrayIdentifier` constructor is "`__init__(self, fc)`", where `fc` is the context object. It is presumably implemented simply as

```
class ArrayIdentifier(Identifier)
  def __init__(self, fc):
    super().__init__(fc)
```

(or equivalent), because it is not meant to actually initialise anything. (As that would not be useful for users extending the library.)

When not extending the library, users create `ArrayIdentifier` objects by running methods of `FTILContext`. Below is the list of such methods available.

Note 5.2. Functions are described here as though they are overloaded. Python does not support overloading, but it is possible to create the same effect using parameter numbers and parameter types for disambiguation. There are several standard ways of how to accomplish this. The presentation here is meant for clarity, rather than as an endorsement of any specific implementation choice.

- `fc.array(typecode, length = 0)` — Creates an `fc.array` of the specified type and length, with all-zero values. (Each of the `ftillite` element types has a natural zero value. For `bytearray` this is the all-zeroes array of the appropriate size; otherwise it is the neutral element for the type's "`+`" operation.) The "`length`" parameter may be an integer or a singleton `fc.array` of integers.
- `fc.array(typecode, length, value)` — The scalar `value` is used to initialise all elements in an `fc.array` of the specified type and length. The "`length`" parameter may be an integer or a singleton `fc.array` of integers. The `value` parameter can be a singleton `fc.array` of any type (or promoted from a Python native) as long as this type can be converted to the destination type specified by `typecode`.
- `fc.array(typecode, values)` – Initialisation from a native Python list. Is available if `typecode` is `'i'` or `'f'`.

5.2.2 Methods

Note 5.3. Some of the methods listed below refer to summation of elements. Most element types in `ftillite` have a "`+`" operation, which is what is used here. The one exception is `'b'`, where we will use logical OR in every case, instead.

The `ftillite` library takes its cue from Python itself, in exposing as much functionality as possible in the form of methods rather than of functions, whilst providing functions/operators with a convenient interface that merely call these underlying methods. This, in a dynamically-typed language such as Python, is a design pattern that allows one to extend functionality into new types, while retaining the convenient syntax used by basic types. Many methods in the following list are such methods, working like native Python "magic methods" and not intended to be called directly at all. We list separately the function calls triggering them. This triggering code can be a single-line pure-Python function.

Note 5.4. In Python, magic methods typically have names that begin and end in double underscores to signal to the user that they are not for direct use. While we mostly use only magic methods that Python already defines, and while most of our methods (magic or not) do not use the underscore convention, in a few places we introduce some methods that very clearly were never meant to be invoked directly by a user, and in these cases we use Python's underscore convention to signal this.

`ArrayIdentifier` inherits the implementations of the following methods from `Identifier`:

- `.context()`,
- `.scope()`,
- `.width()`,
- `.auxdb_read(query)`, and
- `.typecode()`.

Of these, "`.typecode()`" must be overwritten by the concrete, standard `ArrayIdentifier` sub-types (though not in `ArrayIdentifier` itself), "`.width()`" and "`.auxdb_read(query)`" can both be overwritten here with implementations more efficient than the default. In particular, "`.width()`" can simply be implemented as "`return 1`".

Inherited methods that are abstract in `Identifier` but implemented here:

`.transmitter()`: Creates and returns a new `ArrayTransmitter` object. This `ArrayTransmitter` can only be used to transmit `ArrayIdentifier`s of the same typecode as the one that created it. Possible implementation: "`return ArrayTransmitter(self)`".

`.stub()`: "`return self.context().array(self.typecode())`".

`.flatten()`: The purpose of this method is to prepare a type to be used as a key to a `listmap`. In the case of `fc.array` it is merely a method that returns "`[self]`", i.e. the left-hand side object as an element in a Python list of length 1.

`.copy()`: Can be implemented as "`return self[:]`".

Inherited methods that are still abstract here, but implemented in concrete sub-types:

`.__setitem__(key, value)`: Allows element and slice in-place assignment. The `key` can be a Python key (or slice), or it can be an integer `fc.array` with index values. The `value` is an array that must have exactly the appropriate length (up to broadcasting, as explained in Section 3.2.4). Its typecode must be the correct typecode for the destination, but it can be promoted as explained in Section 3.2.3. If `key` is an `array`, all of its values must be between zero and the length of the left-hand side `array`, or an exception is raised. It is not allowed for the `key` to have multiple values that are identical. If this happens, the result is implementation dependent. If `key` is `':'` (meaning "`slice(None, None, None)`"), the entire contents of the array are overwritten, and the new values do not need to have the same length as the old ones. (They do, however, need to be of the same type.)

`.unflatten(data)`: The inverse operation from "`.flatten()`". Up to type checking, this is simply

```
self[:] = data[0]
return self
```

`.sametype(other)`: For standard, concrete, `ArrayIdentifier` subtypes, this is equivalent to:

```
return type(other) is EXPECTED_TYPE and self.typecode() == other.typecode()
```

where "`EXPECTED_TYPE`" is the type of `self` if `self` is an `ftillite` type, but may be different if a user inherits from the standard types.

The typecode check can be omitted for all subtypes except `BytearrayArrayIdentifier`.

`ArrayIdentifier`-specific methods:

`.promote(typecode)`: This is a helper method for implementing automatic type promotion in library extensions. The parameter `typecode` is a Python string. The return value is a tuple, `(v, is_copy)`. If `typecode` is identical to the typecode of the left-hand side object, the object itself is returned as `v` and the Python Boolean value `False` is returned as `is_copy`. If the left-hand side is an `IntArrayIdentifier` and the typecode is either `'f'` or `'I'`, a copy converted to the desired type is returned as `v`, and `is_copy` is `True`. Otherwise, a `TypeError` exception is raised. This method has a default implementation here, but will be overwritten in `IntArrayIdentifier`.

`.astype(typecode)`: Abstract here. Returns a new array of the same length, but with all values converted to the given typecode. Throws an exception if any value cannot be converted to its target type in a non-lossy manner. See Section 5.2.3 for the list of supported conversions. If `typecode` is the original type, the operation works like a copy constructor, performing deep copying.

`.len()`: Abstract here. Returns a singleton integer array, its value in each node being the length of the array in that node.

`.set_length(new_length)`: Abstract here. The parameter `new_length` is a singleton integer array (or a Python integer). The `array` changes its length. If the new length is shorter than the existing length, the array is truncated. If it is longer, zero elements are added. (See Section 5.2.1 for a discussion of what the zero element of each type is.) The method `set_length` should be handled in-place as much as possible, meaning that small length changes to very large arrays should not trigger excessive data copying.

`.empty()`: Equivalent to "`.set_length(0)`".

`.__getitem__(key)`: Abstract here. Allows element and slice access to the `array`. The `key` can be a Python key (or slice), or it can be an integer `fc.array` with index values. The result is a new array, of the same typecode, that contains only the elements from the requested positions, in their requested order. If `key` is an `fc.array`, its values must be nonnegative and less than the length of the left-hand side `array`, or an exception is raised. (If a Python slice is sent, negative values mean offsets from the end of the list, as usual.) To copy an entire `fc.array`, use ":" as the slice definition.

`.lookup(key, default = None)`: Abstract here. Identical to `__getitem__`, except that out-of-range `key` values cause the `default` value to be returned, and no exception is ever raised. If no `default` is explicitly given, it is the type's zero value. (See Section 5.2.1.)

`.reduce_sum(key, value)`: Abstract here. Identical to `__setitem__`, except that identical key values are allowed. The value assigned to the left-hand side `array` is the sum of all `values` matching a given `key`. If any `array` element is not matched by any `key`, its value remains unchanged. In this method, `key` can be a Python integer or an `fc.array`, but cannot be a Python slice.

`.reduce_isum(key, value)`: Abstract here. Identical to `reduce_sum`, except that the new value is added onto the existing value of the left-hand side `array`, rather than replacing it entirely.

`.__delitem__(key)`: Abstract here. Allows element and slice deletion. The `key` can be a Python key (or slice), or it can be an integer `fc.array` with index values. Other elements are moved to compensate.

`.contains(items)`: Abstract here. The parameter `items` is an `ArrayIdentifier` of the same type as the left-hand side (after promotion). Returns an `IntArrayIdentifier` of the same length as `items`. Its value in each position is 1 if the item in the respective position in `items` is contained in the left-hand side `array` and 0 otherwise.

`.cumsum()`: Abstract here. The return value is an `array` of the same type and length as the original `array`, such that its value at each position i is the sum of all the original array elements in positions $[0, i]$.

`.index()`: Abstract here. The left-hand side can be an `array` of any type. The return value is an `fc.array` of typecode `'i'`, whose contents are the positions in which the value of the left-hand side `array` are nonzero. (The order of the values in the returned `array` is arbitrary.)

`.serialise()`: Abstract here. Converts the data from any `fc.array` type into an `fc.array` of `bytearray`. The choice of bit representation is arbitrary, as long as the function is invertible. In particular, it is possible to serialise `array`s of floats, which cannot simply be converted to the `bytearray` format.

`.deserialise(data)`: Abstract here. Reinterprets the `BytearrayArrayIdentifier` given in `data` as an `ArrayIdentifier` of the current object's type and uses it to replace the previous payload of the object. If `data` was originally created by a call to "`.serialise`" beginning with an object of the current type and typecode, the process is guaranteed to yield back the original value. If it was created through other means or from a different type, results are implementation-dependent, and may cause an exception to be raised.

`.broadcast_value(length):` Concrete here, but requires a non-Python implementation because it involves communication. This is a helper method used in automatic value broadcasting. (See Section 3.2.4.) Length is a singleton integer `array`. It returns a tuple `(v, is_copy)`. The semantics of the method are as follows. If `length` equals the length of the left-hand side in every node, return the left-hand side itself (with no copying) as `v`, and the Python Boolean `False` as `is_copy`. Otherwise, if the left-hand side is a singleton (in every node), return a copy that has in every node `length` copies of the same original value in `v`, and `True` in `is_copy`. If the left-hand side is not a singleton in at least one node, raise a `ValueError` exception.

`.__mux__(conditional, iffalse)` **and** `.__rmux__(conditional, iftrue):` Abstract here. Magic methods that implement the "`mux`" functionality. See Section 6.4.

`.__eq__(other)` **and** `.__ne__(other):` Abstract here. Expect `other` to be, after promotion and broadcasting, of the same type, typecode and length as the left-hand side. These standard Python magic methods are used to return an `IntArrayIdentifier` of the same length as the arguments, which is 0 or 1 in each position i depending on whether the left-hand side and `other` equal each other or not in this position. It is expected that all concrete `ArrayIdentifier`s support this operation. In particular, as discussed in Section 7, all standard `ftillite ArrayIdentifier`s support them.

Note 5.5. If Python had been a language that requires interfaces, then "`.serialise()`" and "`.deserialise()`" would have been methods of the interface class "`SerialisableIdentifier`", located in the hierarchy between "`Identifier`" and "`ArrayIdentifier`". There will certainly be other classes derived from `Identifier` that will implement these two methods. Given that Python is "duck-typed", we decided not to introduce this intermediate class.

Some extenders of the library may not want to provide "`.serialise()`" and "`.deserialise()`" methods for their `ArrayIdentifier`-derived types.

Note 5.6. The methods `reduce_sum`, `reduce_isum` and `cumsum` appear in this list because all built-in scalar types support summation (with `bytearray` interpreting this as a logical OR). It is not, however, an intrinsic part of what it means to be an `ArrayIdentifier`. Indeed, the example of `Pair` in Section 8 demonstrates that one can be an `ArrayIdentifier` without any arithmetic operations at all.

Again, if Python had required strict adherence to interfaces, the solution would have been to create an appropriate hierarchy, with these methods appearing only in a subtype of the `ArrayIdentifier` type, e.g. a new "`SummableArrayIdentifier`" type.

5.2.3 Typecode conversions

Typecode conversions for `fc.array`s are mostly done using the "`.astype(typecode)`" method introduced in Section 5.2.2.

Table 1 summarises the available conversions, by typecodes.

Table 1: The allowed type conversions.

From \ To	i	f	I	E	b
i	•	✓	✓	✓	✓
f		•			
I	✓		•	✓	✓
E				•	
b	✓		✓		

Type conversions from a type to itself act as a simple copy constructor, performing deep copying. The entry in Table 1 that relates to type conversions from type `'b'` to itself indicates conversions that involve a change of size.

The conversions that are labelled in the table as allowed may still fail, raising an `OverflowError` exception, if the specific values to be converted do not fit in the target type (including any given size specification). This is the case with

- Conversion from an `Ed25519Int` to an integer,
- Conversion from an integer to a float, where the integer is too large to be represented as a float without loss, and
- Conversion to and from `bytearray`, where the destination type does not have enough bits to store the relevant information.

When converting between integer (including `Ed25519Int`) and `bytearray` types, a particular `bytearray` element `ByteValue`, with its bytes represented as

`ByteValue[0]`,..., `ByteValue[size-1]`,

is considered the `bytearray` equivalent to an integer `IntValue` (of any integer type), if

$$IntValue = \sum_{i=0}^{size\text{-}1} ByteValue[i] \times 256^i.$$

The `fc.array` type also includes the methods `.serialise()` and `.deserialise()` that allow converting arrays of any type to and from an `fc.array` of `bytearray`s. One can think of this like a C++ "`reinterpret_cast`", in the sense that no specific meaning is given to the byte values of the `bytearray`. The only requirement is that if a certain `bytearray` was created by a call to `.serialise()`, a subsequent call to `.deserialise()` with the original typecode as its parameter will restore the original values.

Serialisation/deserialisation can happen only if the `bytearray` has exactly the appropriate size. This should be 8 for `integer` and `float`, 32 for `Ed25519Int`, and 64 for `Ed25519`. How the `bytearray` stores the information is immaterial, as long as the following conditions are met:

1. The conversion is deterministic and invertible,
2. The same conversion is used on all nodes, and
3. The choice of storage format facilitates speed of execution.

If, when attempting to deserialise a `BytearrayArrayIdentifier`, any element's value is not the serialised image of any value in the target type, results are undefined. An exception may or may not be triggered.

In addition to the above, standard `ftillite` concrete types that inherit from `ArrayIdentifier` also support, where appropriate, the following functionality, further explained below, that translates between array types.

- `ceil()`,
- `floor()`,
- `round()`,
- `fl.nearest()`,
- `fl.ceil()`,
- `fl.floor()`,
- `fl.round()`,
- `.byteproject(size, mapping)` and `.concat(x)` (for `BytearrayArrayIdentifier`s only),

- `.ed_folded()` and `.ed_affine()` (for `Ed25519ArrayIdentifier`s only), and
- `.ed_folded_project()` and `.ed_affine_project()` (for `BytearrayArrayIdentifier`s only).

Conversion from float to int will not happen using the "`astype`" method. Instead, `ftillite` will support "`ceil()`", "`floor()`" and "`round()`" functions. Conversion from int to float can also be done using the "`fl.nearest()`" function, and this will always succeed, even when the conversion is lossy. For consistency of interface, `ftillite` will also support "`fl.ceil()`", "`fl.floor()`" and "`fl.round()`". These global functions are all wrappers for magic methods. See Section 7 for details.

The above functions all take a single `fc.array` as a parameter and return a single `fc.array`.

Conversion between `bytearray`s of different sizes can be done using the method

```
.byteproject(size, mapping)
```

Here, the left-hand side ("`self`") is the original `BytearrayArrayIdentifier`, `size` is the destination `bytearray` size, and `mapping` is a Python dictionary, such that if the key-value pair (i, j) is in the mapping, then byte j of the original is stored in byte i of the returned value. Any byte not mapped by any key-value pair is returned as zero. The values of i must be between 0 and `size-1` and the values of j must be between 0 and `self.size()-1` or this is an error.

Arrays also supports the following cryptographic conversion methods.

`.ed_folded()`: The left-hand side is an `fc.array` of `Ed25519`. The return value is an `fc.array` of typecode `'b32'` that stores the information of each `Ed25519` (uniquely) in folded representation.

`.ed_affine()`: The left-hand side is an `fc.array` of `Ed25519`. The return value is an `fc.array` of typecode `'b64'` that stores the information of each `Ed25519` (uniquely) in affine representation.

`fc.ed_folded_project()`: Performs the conversion from folded to projective representation. The left-hand side is an `fc.array` of typecode `'b32'` and the return type is an `fc.array` of typecode `'E'`.

`fc.ed_affine_project()`: Performs the conversion from affine to projective representation. The left-hand side is an `fc.array` of typecode `'b64'` and the return type is an `fc.array` of typecode `'E'`.

The result of `array.ed_folded().ed_folded_project()` may not be bit-for-bit identical to the original, but should have the same `Ed25519` value. In other words:

```
# Given any array of Ed25519
edint_array = fc.randomarray('I', 1000, False)
ed_array = edint_array.astype('E')
restored_array = ed_array.ed_folded().ed_folded_project()

# The restored array is guaranteed to equal the original...
value_identical = (ed_array == restored_array) # This will be all '1's.
fl.verify(value_identical)

# ... but may not be bit-identical to it.
byte_orig_array = ed_array.serialise()
byte_restored_array = restored_array.serialise()
bit_identical = (byte_orig_array == byte_restored_array) # May contain '0's.
```

The same goes for `array.ed_affine().ed_affine_project()`.

5.2.4 `ArrayIdentifier` sub-types

Additionally, the following methods are only for `BytearrayArrayIdentifier`:

`.size()`: Returns the array's size, as a Python integer.

`.ed_folded_project()`, `.ed_affine_project()` **and** `.byteproject(size, mapping)`: See Section 5.2.3.

`.concat(x)`: Where the left-hand side has typecode "b*X*" and the `x` argument is a `Bytearray` array of typecode "b*Y*", the result is a `BytearrayArrayIdentifier` of typecode "b*(X + Y)*" that contains the concatenation of the two bytearrays.

The following method is only for `Ed25519ArrayIdentifier`:

`.ed_folded()` **and** `.ed_affine()`: See Section 5.2.3.

The following methods are only for `IntArrayIdentifier`s and `FloatArrayIdentifier`s:

`.reduce_max(key, value)`: Identical to `reduce_sum`, except that the result is the maximum value rather than the sum.

`.reduce_imax(key, value)`: Identical to `reduce_max`, except that the old value of the left-hand side array participates in the maximum calculation.

`.reduce_min(key, value)`: Identical to `reduce_max`, but with minimum instead of maximum.

`.reduce_imin(key, value)`: Identical to `reduce_imax`, but with minimum instead of maximum.

`.sorted()`: Returns a sorted version of the array.

`.index_sorted(index = None)`: The `index` parameter, if given, should be an integer array of the same length as the left-hand side. The return value is `self.context().arange(self.len())`, but sorted according to ascending order of (l, i), where l are the values of the left-hand side, i are the values of the `index`, and `self` is the left-hand side array. If no `index` is given, it is taken to be `self.context().arange(self.len())`.

`.__iter__()` **and** `.__len__()`: Allow conversion to a Python list, where the list elements are either Python `int`s or Python `float`s, depending on the `ArrayIdentifier` subtype. When conversion is done, only the array's value at the coordinator node is converted. The array must have the coordinator node in its scope or an exception is raised. The method "`.__len__()`", specifically, returns as a Python integer the length of the array at the coordinator node, or raises an exception if it is not defined there. This is not to be confused with the method "`.len()`", which is available for all `ArrayIdentifier` subtypes.

The following method is only for `IntArrayIdentifier`s:

`.__nonzero__()`: This magic method is called when an object is evaluated as a Boolean. The coordinator node must be in the scope of the `IntArrayIdentifier` or an exception is raised. If this prerequisite is met, the `array` evaluates as `True` if all its values on the coordinator node are nonzero, and `False` otherwise.

Note 5.7. It is in general in `ftillite` not allowed to convert to Python (i.e., local) variables any information not exposed to the coordinator node.

In Section 7, we list the operators supported by the various `ArrayIdentifier` subtypes. These operators are mainly supported through Python's magic methods, so require implementing additional methods, beyond those listed here.

5.3 The `listmap` type

An `fc.listmap` is a data structure that maps from a set of distinct keys to the set of consecutive integers beginning with zero. Each `fc.listmap` has a particular type of key, which is specified for it at construction time by means of a typecode. The typecode is a string that concatenates multiple `fc.array` typecodes, and is interpreted as a tuple of values containing these types, in their given order. All `fc.array` typecodes are eligible as typecodes for `listmap` key tuple elements, except for the typecode `'E'`.

For example, a `listmap` can be initialised with the typecode `'fi'`, and this indicates that the key type of the `listmap` is a tuple containing one floating point value and one integer.

For example, such a `listmap` may contain the following mapping in a given node:

```
{(1.384e-14, 13059871895) : 3, (0.349114, 5082190751) : 0,
    (7.243e09, 768719571) : 2, (0.0000451, 131231312659) : 1}
```

Notably, the values mapped to by the `listmap` are necessarily the consecutive integers between zero and the mapping's size.

Typically, when working with `listmap`s, one communicates more than one key and more than one value at a time. To specify multiple keys, one uses a Python list whose elements are `fc.array`s of the appropriate type. To specify the values, we use an `fc.array` of integers. For example, if the scope of the `listmap` above is the coordinator node, the `listmap` can have its keys communicated as

```
[fc.array("f", [1.384e-14, 0.349114, 7.243e09, 0.0000451]),
 fc.array("i", [13059871895, 13059871895, 768719571, 131231312659])]
```

If these values are queried of the `listmap` in this order, the returned values will be in the form

```
fc.array("i", [3, 0, 2, 1])
```

FTIL Note 5.4. The `listmap` type is a central type in simulating FTIL dictionaries. The idea is that if we wish to simulate a `dict<A,B>`, we first break up each of `A` and `B` to their atomic types and store them separately as `fc.array`s. We then use the `listmap` in order to map the keys from `A` into index positions, upon which we can look up these index positions in the `fc.array`s of `B` in order to retrieve values.

5.3.1 Constructors

We do not actually list constructors in this section, because these are never used by the user. Below is a listing of the `FTILContext` methods used to generate new general `ListMapIdentifier`s.

Note 5.8. Functions are described here as though they are overloaded. Python does not support overloading, but it is possible to create the same effect using parameter numbers and parameter types for disambiguation. There are several standard ways of how to accomplish this. The presentation here is meant for clarity, rather than as an endorsement of any specific implementation choice.

- `fc.listmap(typecodes)` — Given a string concatenating `fc.array` typecodes, creates an empty listmap of the relevant type. Typecodes may include any `fc.array` typecode except for `'E'`, including having size specifications. White space between typecodes is optional, but no whitespace is allowed within a typecode.
- `fc.listmap(arrays, order = "any")` — Given a Python list in the parameter "`arrays`", the parameters of the list are first broadcast (See Section 3.2.4), then, once they are all `fc.array`s of equal length, the `listmap` is initialised such that each index i of the `arrays` corresponds to a single key of the `listmap`, composed of the tuple that is all values at index i of each base array. (The interface does not require any such tuple to be materialised, however.) The ordering of the `listmap` values associated with the individual keys is determined

by the Python string parameter `order`. If it is `"any"`, the index value of each element in the input is arbitrary. If `order` is `"rnd"`, the order of the map values is a cryptographically-strong random. If it is `"pos"`, the index position of each element in the original arrays is its `listmap` value. This latter option requires all input keys to be unique or an exception is raised. In all cases, the mapping is created independently in each node in the current execution scope, based on the value of the individual `array`s on each node. Note that if `order` is not `"pos"`, input tuples may not be unique. However, the final `listmap` will store in every case only a single value for each unique input key.

- `fc.listmap(typecodes, arrays, order = "any")` — Same as above, but in addition to being broadcast, the `fc.array` values are converted to the types listed in `typecodes`. This follows standard conversion rules (See Section 5.2.3), and, in particular, may throw exceptions if conversion is not possible.

5.3.2 Methods

`ListMapIdentifier` inherits the implementations of the following methods from `Identifier`:

- `.context()`,
- `.scope()`,
- `.width()`,
- `.auxdb_read(query)`, and
- `.typecode()`.

The following methods are abstract in `Identifier` but implemented here.

`.transmitter()`: A helper method. Creates and returns a new `Transmitter` object. This `Transmitter` can only be used to transmit `fc.listmap`s of the same key typecodes as the one that created it. Possible implementation:

```
return ListMapTransmitter(self)
```

`.stub()`: "`return self.context().listmap(self.typecode())`".

`.sametype(other)`: Equivalent to

```
return type(other) is ListMapIdentifier and \
         self.typecode() == other.typecode()
```

`.flatten()`: Equivalent to "`return self.keys()`".

`.unflatten(data)`: Equivalent to

```
self[:] = self.context().listmap(data, "pos")
return self
```

`.copy()`: Returns a deep copy of the object.

`.__setitem__(key, newlistmap)`: Raises an exception unless `key` is `slice(None, None, None)`, i.e. unless it is `':'`. If it is, `newlistmap` is expected to be a `ListMapIdentifier` of the same type (typecode) as `self`, the left-hand side variable. All of the data from `self` is discarded and replaced by the data of `newlistmap`.

Methods unique to `ListMapIdentifier`:

`.len()`: Returns a singleton integer array, its value in each node being the number of keys in the listmap in that node. Not to be confused with "`.__len__()`", described above.

`.keys()`: Returns the keys of the listmap as a Python list of `fc.array`s. The order of the keys in the `fc.array`s is their index order. (Note that it is the responsibility of the user to sort the elements of these `fc.array`s if one wants to scrub any unwanted information being carried by the sorting order.)

`.todict()`: A `listmap` can be converted to a Python dictionary using the "`todict`" method. When this is done, only the values stored at the coordinator node are used. The key types, in this case, must only include values of base typecode `'i'` and `'f'`, and the resulting keys are Python tuples with integer and float member values, accordingly.

`.__getitem__(key)`: Allows element and slice access to the `listmap`. The `key` is a Python list. This list is first broadcast (as explained in Section 3.2.4), making it a list of `fc.array` elements, all with the same length. The type of elements in the list must correspond exactly to the expected key types (subject to promotion, as explained in Section 3.2.3). If the elements of `key` do not match the expected types, this raises a `TypeError` exception. The result of the call is an `IntegerArrayIdentifier` that contains the values for the respective keys in each position. The call raises a `KeyError` exception if a key is not found.

`.lookup(key, default = -1)`: Same as above, but if a key is not found, `default` is returned. This method does not raise `KeyError` exceptions.

`.contains(items)`: The value of `items` is of the same type as `key` above. The method returns an integer `array` of the same length as the arrays of `items` (after promotion and broadcasting). Its value in each position is 1 if the item in the respective position in `items` is contained in the `listmap`'s keys and 0 otherwise.

`.add_items(items)`: The parameter `items` is either as above or is an `fc.listmap` of the same `typecode`. If it is a `listmap`, the input is taken as though it was `items.keys()`. The items in `items` are expected not to be keys of the left-hand side `listmap`. If any are, an exception is raised. The items are added with the previously-unused `[old_length, new_length)` values to the `fc.listmap`. This is done in-place, in the sense that there is no copying of existing items. The method's return value is a tuple `(new_keys, new_values)`, where `new_keys` is the keys added, presented as a list of `fc.array`s in the usual format for keys, and `new_values` is an integer `fc.array` of the same length. The `new_keys` list contains the new keys that have been added to the `listmap`, and `new_values` contains for each new key the index value that the `listmap` maps it to. The values in `new_keys` are the same as in `items`, except that any repeating keys in `items` are removed.

`.merge_items(items)`: Same as `add_items`, except that items that already exist in the `listmap` are silently ignored and no exception is ever raised.

`.remove_items(items)`: Takes parameters of the same kind as `add_items`, but these items must all be keys of the left-hand side `fc.listmap` or an exception is raised. The items are removed from the keys of the `listmap`, and keys that map to values in `[new_length, old_length)` are moved to new values to compensate. This is done in-place, in the sense that there is no copying of unaffected items. The method's return value is a tuple `(moved_keys, old_values, new_values)`, where `moved_keys` are keys in the standard format, and for each key `old_values` and `new_values` list for it what its previous index value was and what its new index value is, respectively. The return values `old_values` and `new_values` are integer `fc.array`s. Keys that were removed or that were not altered are not returned in this tuple. Keys in `items` may not repeat.

`.discard_items(items)`: Same as `remove_items`, except that any items that are not originally keys of the list are silently ignored, keys may repeat, and no exception is ever raised.

`.intersect_items(items)`: Takes parameters of the same kind as `add_items` (which may or may not be keys of the left-hand side `fc.listmap` and may or may not include repetitions). Returns a new `fc.listmap` whose keys are the intersection of the original `listmap`'s keys and the items in `items`.

6 `ftillite` functions

Python `ftillite` library functions accept parameters, some of which are `ftillite Identifier`s, and manipulate the back-end objects identified by these `Identifier`s. Some functions require communication (and these will be specifically highlighted) but most are embarrassingly parallel, and merely run the same operation on every node in the current execution scope.

Almost all `ftillite` functions run on the nodes specified by "`fc.scope()`". Unless stated otherwise, it is an error to run a function on an object beyond that object's scope.

Functions that create and return new `fc.array` and `fc.listmap` objects create these, unless otherwise stated, with their scope being `fc.scope()`.

In order to allow the Python programmer to extend the library (See Section 8), much of the functionality that is exposed to the user as global functions is implemented by these global functions as simple wrappers around calls to methods of the underlying objects. This is analogous to Python's use of "magic methods".

6.1 Preempting execution

`fl.verify(condition)`: The parameter `condition` is an `IntArrayIdentifier`. If any of its elements equals zero in any node that is in the execution scope, this raises an `AssertionError` exception.

6.2 Inter-node communication

`fl.transmit(dictionary)`: This is a convenience function. The parameter `dictionary` is a Python dictionary whose keys are `Node`s. If the dictionary is empty, an exception is raised. Otherwise, the function picks an arbitrary dictionary value, `x`, and runs

```
t = x.transmitter()
return t.transmit(dictionary)
```

Note 6.1. This is `ftillite`'s main function that requires communication. It is also a function whose parameters may not be defined across the entire current execution scope. See Section 2.9 for details.

6.3 Cryptographic functions

Note 6.2. As described in other FTIL documents, the requirement is really for secure cryptographic primitives of various types, not for specific algorithms. We suggest here particular algorithms for concreteness and in order to demonstrate the full syntax.

`fl.sha3_256(data)`: Takes a `BytearrayArrayIdentifier` (of any size) and returns an `fc.array` of the same length and of typecode `'b32'` with the SHA3-256 hash of the input. (Other secure hash functions can be implemented instead.)

`fl.ecdsa256_sign(data, priv_key)`: The parameter `data` is a `BytearrayArrayIdentifier` (of any size). The parameter `priv_key` is a singleton ECDSA private key, as returned from `fc.ecdsa256_keygen()`. The return value is a `BytearrayArrayIdentifier` of the same length as `data` and of appropriate size. Each element in the result is the digital signature of the corresponding element in `data`.

`fl.ecdsa256_verify(data, signature, pub_key)`: With `data` being as above, `signature` being of the same type as a return value from `fl.ecdsa256_sign` and the same length as data (after broadcasting), and `pub_key` being a (singleton) public key as returned by the key generating method `fc.ecdsa256_keygen`, the function returns an `IntArrayIdentifier` of the same length, such that an element in the result is 1 if the corresponding `data` element matches the corresponding `signature` given the public key and 0 otherwise.

`fl.aes256_encrypt(data, key)`: The parameter `data` is an `fc.array` of typecode `'b16'` and `key` is a singleton `fc.array` of typecode `'b32'`. The result is an `fc.array` of the same length and type as `data`, where each element of `data` was individually and independently encrypted using AES (i.e., using ECB mode). (Other secure block ciphers can be used instead.)

`fl.aes256_decrypt(data, key)`: Same parameter and return types as `aes256_encrypt`. This function is the inverse of the previous function.

`fl.grain128aeadv2(key, iv, size, length)`: The parameters `key` and `iv` are both singleton `bytearrays` of the appropriate size to be a Grain-128AEADv2 key and initialisation vector, respectively. The parameter `size` is a Python integer, and `length` is an `IntArrayIdentifier` singleton. The return value is a `BytearrayArrayIdentifier` of the specified length and size, whose contents have been generated by the Grain-128AEADv2 stream cipher. (Other secure stream ciphers can be used instead.)

`fl.rsa3072_encrypt(data, pub_key)`: The parameter `data` is a `BytearrayArrayIdentifier` of the appropriate size. The parameter `pub_key` is a singleton RSA public key, as returned from `fc.rsa3072_keygen()`. The return value is a `BytearrayArrayIdentifier` of the same length as `data` and of an appropriate size. Each element in the result is the ciphertext of the corresponding element in `data`.

`fl.rsa3072_decrypt(data, priv_key)`: The parameter `data` is a `BytearrayArrayIdentifier` of the appropriate size. The parameter `priv_key` is a singleton RSA private key, as returned from `fc.rsa3072_keygen()`. The return value is a `BytearrayArrayIdentifier` of the same length as `data` and of an appropriate size. Each element in the result is the decryption of the corresponding ciphertext element in `data`.

6.4 Ternary operator

`fl.mux(conditional, iftrue, iffalse)`: This is a convenience function. Its normal functionality is to take three `fc.arrays` of the same length (after broadcasting), of which the first is of integer type, and the other two are of any but equal types (after promotion). The result is an `fc.array` of the same length, of the same type as `iftrue` and `iffalse`, whose value equals the value of `iftrue` at any index where `condition` is nonzero, and the value of `iffalse` where `conditional` is zero. However, `fl.mux` does not actually compute any of this. Instead, it does the following:

```
rc = NotImplemented
if hasattr(iftrue, "__mux__"):
  rc = iftrue.__mux__(conditional, iffalse)
if rc != NotImplemented:
  return rc
elif hasattr(iffalse, "__rmux__"):
  rc = iffalse.__rmux__(conditional, iftrue)
if rc != NotImplemented:
  return rc
elif set([type(iftrue), type(iffalse)]).issubset(set([int, float])):
  return (conditional != 0) * iftrue + (conditional == 0) * iffalse
else:
```

```
raise TypeError("Operands do not support mux function.")
```

The idea here is twofold:

1. The user can extend the "`mux`" functionality using their own types, and
2. Value broadcasting is delayed into the `__mux__` and `__rmux__` methods to allow said user to provide a better solution than just automatic value broadcasting. (Although, if value broadcasting is desired, it can easily be implemented by the user, by utilising the helper functions "`fl.calc_broadcast_length`" and "`.broadcast_value`".)

6.5 Miscellaneous functions

`fl.get_context(params):` The argument is a Python list. Return "`x.context()`" for any element `x` in `params` that is a `ManagedObject`. If none are `ManagedObject`s, raise a `TypeError` exception. Before a value, `fc`, is returned, the function also runs "`fc.verify_context(params)`".

6.6 Mass operations

FTIL Note 6.1. This section follows Section 3.12 of [1].

The `ftillite` library includes a block command that works as a "hint" for the underlying implementation, though it requires no functional support. The idea is that this hint allows for much more efficient execution.

The command has the following syntax:

```
with fl.massop():
  # Operations within the inner context are taken to be part of a single
  # mass operation.
```

The idea is that inside the "`fl.massop()`" scope, multiple operations may occur that may create and delete variables rapidly. Only those new variables that continue to exist when the scope ends are actually required to be persisted at any point. All others can safely be considered intermediate variables. In implementation terms, such variables can be kept in system caches and never truly "materialised".

Multiple "`fl.massop()`" scopes can be nested.

Note 6.3. The present document does not advocate for any particular implementation choices, but "`massop()`" can, in principle, be implemented analogously to the "`on()`" implementation described in Section A.3.

Note 6.4. At the present time it is unclear how this hint will be used by the system, in the PoC or otherwise. As a result, it is unclear what additional information the "`massop()`" hint should provide in order to be useful. For this reason, at the present time, this operation is described as receiving no arguments at all. Once it becomes clear what additional information is required, this can be added as arguments. If, for example, the system needs to be given the identity of the `FTILContext` object, this can be passed as a parameter to the "`massop()`" function.

7 `ftillite` operators

Table 2 describes which operators are supported by which `fc.array` type for the execution of element-by-element operations. All inputs are `fc.array`s of the same length, up to automatic broadcasting (regarding which see Section 3.2.4). The typecode listed on the table is the typecode of the `fc.array` object that should implement the magic method listed. Usually, this means that it is the left-most operand of the operation. The notes in the table discuss the typecodes of the other operands. Where not otherwise noted, the return value is assumed to have the same typecode as the method-invoking object, after any promotion is applied.

Table 2: Supported operations on `fc.arrays`.

Magic method \ Operands' type	`i`	`f`	`I`	`E`	`b`	Comments
`__eq__`	✓	✓	✓	✓	✓	See Notes 1, 2, 3.
`__ne__`	✓	✓	✓	✓	✓	See Notes 1, 2, 3.
`__lt__`	✓	✓				See Notes 1, 2.
`__gt__`	✓	✓				See Notes 1, 2.
`__le__`	✓	✓				See Notes 1, 2.
`__ge__`	✓	✓				See Notes 1, 2.
`__pos__`	✓	✓	✓	✓		
`__neg__`	✓	✓	✓	✓		See Note 4.
`__abs__`	✓	✓				
`__floor__`		✓				See Note 5.
`__ceil__`		✓				See Note 5.
`__round__`		✓				See Note 5.
`__add__`	✓	✓	✓	✓		See Note 1.
`__iadd__`	✓	✓	✓	✓		See Notes 1, 6.
`__radd__`	✓	✓	✓			See Note 1.
`__sub__`	✓	✓	✓	✓		See Note 1.
`__isub__`	✓	✓	✓	✓		See Notes 1, 6.
`__rsub__`	✓	✓	✓			See Note 1.
`__mul__`	✓	✓	✓	✓		See Notes 1, 7.
`__imul__`	✓	✓	✓	✓		See Notes 1, 6, 7.
`__rmul__`	✓	✓	✓	✓		See Notes 1, 7.
`__floordiv__`	✓		✓			See Notes 1, 8.
`__ifloordiv__`	✓		✓			See Notes 1, 6, 8.
`__rfloordiv__`	✓		✓			See Notes 1, 8.
`__truediv__`	✓	✓	✓			See Notes 1, 8.
`__itruediv__`	✓	✓	✓			See Notes 1, 6, 8.
`__rtruediv__`	✓	✓	✓			See Notes 1, 8.
`__mod__`	✓					See Note 1.
`__imod__`	✓					See Notes 1, 6.
`__rmod__`	✓					See Note 1.
`__divmod__`	✓					See Note 1.
`__rdivmod__`	✓					See Note 1.
`__pow__`	✓	✓	✓			See Note 9.
`__ipow__`	✓	✓	✓			See Note 9.
`__rpow__`	✓					
`__lshift__`	✓				✓	See Note 9.
`__ilshift__`	✓				✓	See Note 9.
`__rlshift__`	✓					
`__rshift__`	✓				✓	See Notes 9, 10.
`__irshift__`	✓				✓	See Notes 9, 10.
`__rrshift__`	✓					See Note 10.
`__and__`	✓				✓	See Note 1.
`__iand__`	✓				✓	See Notes 1, 6.
`__rand__`	✓					See Note 1.
`__or__`	✓				✓	See Note 1.
`__ior__`	✓				✓	See Notes 1, 6.
`__ror__`	✓					See Note 1.
`__xor__`	✓				✓	See Note 1.
`__ixor__`	✓				✓	See Notes 1, 6.

(Continued on next page.)

Magic method \ Operands' type	i	f	I	E	b	Comments
__rxor__	✓					See Note 1.
__invert__	✓				✓	
__nearest__	✓					See Note 11.
__exp__		✓				See Note 11.
__log__		✓				Natural log. See Note 11.
__sin__		✓				See Note 11.
__cos__		✓				See Note 11.

Notes on the table:

1. These operations expect their parameters to have the same type. If they are not, automatic promotion is applied, as described in Section 3.2.3.
2. The return value's elements are 0/1 integers.
3. All `ArrayIdentifier`s, including any user-defined ones, are expected to support equality/inequality operators.
4. For typecodes `'I'` and `'E'`, these are negations in their respective groups.
5. These magic methods are triggered by standard Python functions. For example "`floor(x)`" triggers "`x.__floor__()`". However, for consistency of interface, we will also trigger these same magic methods with custom library functions. For example, in addition to "`floor(x)`", we will also support the interface: "`fl.floor(x)`" to mean the same.
6. The left-hand side, in assignment operations, does not get promoted or broadcast.
7. Products of `'E' * 'I'` and `'I' * 'E'` are allowed, including with promotion from `'i'` to `'I'`. The result is of type `'E'`, for which reason `'E *= I'` is allowed, but not its reverse. `Ed25519ArrayIdentifier`s cannot be arguments in any other type of multiplication.
8. When applied on an `Ed25519IntArrayIdentifier`, "floor division" is integer division whose result is a rounding down to the nearest integer, but "true division" is division under the group's modulo.
9. The right-hand-side must have base typecode `'i'`.
10. When right-shifting a `bytearray`, zeroes are appended on the left. When right-shifting an integer, sign bits.
11. These are not standard magic methods. We will call them, however, from corresponding global library functions. For example, when a user writes "`fl.cos(x)`", this call will invoke "`x.__cos__()`".

Additional notes:

1. Overflows and underflows lead to the appropriate exceptions being raised, as does division by 0. In all cases, if the return type cannot store the returned result, this results in an `OverflowError` exception.
2. Integer division is handled the Python way: the result of division is always rounded towards negative infinity, and `a % b` is defined as $a - b\lfloor a/b \rfloor$.
3. Rounding of half-integers is always upwards.

8 Examples

8.1 Defining a Pair

In an actual implementation, we will want to extend `ftillite` to include various more complex types like account identifiers or struct types, which will inherit from `ArrayIdentifier`. In this example, we will create a relatively simple new construct, namely a `Pair`. Not all functionality of `ArrayIdentifier` will be implemented, in order to keep the example short, but enough will be implemented to support, e.g., pairs of pairs.

```
class Pair(fl.ArrayIdentifier):
  def __init__(self, first, second):
    fc = fl.get_context([first, second])
    super().__init__(fc)
    (first, is_copy1) = fc.promote(first)
    (second, is_copy2) = fc.promote(second)
    length = fc.calc_broadcast_length([first, second])
    (first, is_copy) = first.broadcast_value(length)
    is_copy1 = is_copy1 or is_copy
    (second, is_copy) = second.broadcast_value(length)
    is_copy2 = is_copy2 or is_copy
    if is_copy1:
      self.first = first
    else:
      self.first = first.copy()
    if is_copy2:
      self.second = second
    else:
      self.second = second.copy()

  def transmitter(self):
    return PairTransmitter(self)

  def stub(self):
    return Pair(self.first.stub(), self.second.stub())

  def sametype(self, other):
    return type(other) is Pair and self.first.sametype(other.first) and \
             self.second.sametype(other.second)

  def __eq__(self, other):
    if type(other) is not Pair:
        return 0
    return (self.first == other.first) & (self.second == other.second)

  def __ne__(self, other):
    return 1 - (self == other)

  def flatten(self):
    return self.first.flatten() + self.second.flatten()

  def unflatten(self, data):
    self.first.unflatten(data[:self.first.width()])
    self.second.unflatten(data[self.first.width():])
```

```
def width(self):
  return self.first.width() + self.second.width()

def typecode(self):
  return self.first.typecode() + self.second.typecode()

def copy(self):
  return Pair(self.first, self.second)

def promote(self, typecode):
  if self.typecode() != typecode:
    raise TypeError("Cannot promote a Pair")
  return (self, False)

def astype(self, typecode):
  if self.typecode() != typecode:
    raise TypeError("Cannot convert a Pair")
  return self.copy()

def len(self):
  return self.first.len()

def set_length(self, new_length):
  self.first.set_length(new_length)
  self.second.set_length(new_length)

def empty(self):
  self.first.empty()
  self.second.empty()

def __getitem__(self, key):
  return Pair(self.first[key], self.second[key])

def lookup(self, key, default = None):
  if default is None:
    return Pair(self.first.lookup(key), self.second.lookup(key))
  elif isinstance(default, Pair):
    return Pair(self.first.lookup(key, default.first), \
                self.second.lookup(key, default.second))
  else:
    raise TypeError("Unexpected type for default.")

def __setitem__(self, key, value):
  if not isinstance(value, Pair):
    raise TypeError("Cannot set a Pair to a non-Pair value.")
  self.first[key] = value.first
  self.second[key] = value.second

def __delitem__(self, key):
  del self.first[key]
  del self.second[key]

def contains(self, items):
  fc = self.context()
```

```
    if not self.sametype(items):
      raise TypeError("Type of items does not match type of container.")
    mylistmap = fc.listmap(self.flatten())
    return mylistmap.lookup(items.flatten()) != -1

  def index(self):
    fc = self.context()
    index1 = fc.listmap([self.first.index()])
    index2 = fc.listmap([self.second.index()])
    pair_index = index1.intersect_items(index2)
    return pair_index.keys()

  # We skip here implementation of "serialise" and "deserialise", for brevity.
  def serialise(self):
    raise NotImplementedError("Serialise not yet implemented.")

  def deserialise(self, data):
    raise NotImplementedError("Deserialise not yet implemented.")

  def broadcast_value(self, length):
    (first, is_copy1) = self.first.broadcast_value(length)
    (second, is_copy2) = self.second.broadcast_value(length)
    if not (is_copy1 or is_copy2):
      return (self, False)
    return (Pair(first, second), True)

  def __mux__(self, conditional, iffalse):
    if type(iffalse) is not Pair:
      raise TypeError("Both sides of mux must have same type.")
    return Pair(fl.mux(conditional, self.first, iffalse.first), \
                fl.mux(conditional, self.second, iffalse.second))

  # There is no need to implement __rmux__, because nothing can be promoted to
  # a Pair.


class PairTransmitter(fl.Transmitter):
  def __init__(self, obj):
    super().__init__(obj.context())
    self.first = obj.first.transmitter()
    self.second = obj.second.transmitter()
  def transmit(self, map):
    if type(map) is not dict:
      raise TypeError("Expected dict argument in 'transmit'.")
    self.context().verify_context(map.values())
    if not set(map.keys()).issubset(self.context().scope()):
      raise KeyError("Map keys are out of scope in 'transmit'.")
    for k in map:
      if type(map[k]) is not Pair:
        raise TypeError("Map values incompatible with transmitter.")
      if not map[k].scope().issubset(self.context().scope()):
        raise KeyError("Transmitted values are outside execution scope.")
    t1 = self.first.transmit({n : item.first for n, item in map.items()})
    t2 = self.second.transmit({n : item.second for n, item in map.items()})
```

```
    out_map = {}
    for n in t1.keys():
      with fl.on(t1[n].scope()):
        out_map[n] = Pair(t1[n], t2[n])
    return out_map
```

With this type, we can already store account identifiers as

```
accountid = Pair(accountnum, bsb)
```

or even transactions as

```
tx = Pair(accountid1, accountid2)
```

8.2 Defining a Dict

The definition of `ftillite` does not include sets or dictionaries, but the core of their functionality is delivered by the `ListMap` class. Here, we show how a full dictionary can be built using the existing primitives. This dictionary will map from a generic type A to a generic type B. Note that A and B do not have to be basic `ftillite` types. They can also be extended types. For example, either or both can be `Pair`s.

```
class Dict(fl.Identifier):
  def __init__(self, k, v):
    # The keys must be unique, for valid Dict construction.
    fc = fl.get_context([k, v])
    super().__init__(fc)
    (self.k, is_copy) = fc.promote(k)
    if not is_copy:
      self.k = self.k.copy()
    (self.v, is_copy) = fc.promote(v)
    (self.v, is_copy2) = self.v.broadcast_value(self.k.len())
    if not is_copy and not is_copy2:
      self.v = self.v.copy()
    # The next line verifies that the keys are, indeed, unique.
    self.listmap = fc.listmap(self.k.flatten(), order = "pos")

  def transmitter(self):
    return DictTransmitter(self)

  def stub(self):
    return Dict(self.k.stub(), self.v.stub())

  def sametype(self, other):
    return type(other) is Dict and \
             self.k.sametype(other.k) and self.v.sametype(other.v)

  def flatten(self):
    return self.k.flatten() + self.v.flatten()

  def unflatten(self, data):
    self.k.unflatten(data[:self.k.width()])
    self.v.unflatten(data[self.k.width():])
    self.listmap = self.context().listmap(self.k.flatten(), order = "pos")
```

```
def width(self):
  return self.k.width() + self.v.width()

def typecode(self):
  return self.k.typecode() + self.v.typecode()

def copy(self):
  return Dict(self.k, self.v)

def __setitem__(self, k, v):
  if type(k) is slice:
    if k != slice(None, None, None):
      raise KeyError("Only the slice ':' is supported.")
    self.context().verify_context(v)
    if not self.sametype(v):
      raise TypeError("Incompatible type in assignment.")
    self.k[:] = v.k
    self.v[:] = v.v
    self.listmap = fc.listmap(self.k.flatten(), order = "pos")
  else:
    # Items in "k" must be unique.
    (new_keys, new_values) = self.listmap.merge_items(k.flatten())
    self.k.set_length(self.listmap.len())
    self.v.set_length(self.listmap.len())
    self.v[self.listmap[k.flatten()]] = v # If items in "k" are not unique,
                                          # this fails.
    self.k[new_values] = new_keys
    # A more complete implementation would have had the code wrapped inside a
    # "try" block in order to roll back the change if anything fails.

def _raw_update_key(self, key):
  (new_keys, new_values) = self.listmap.merge_items(key.flatten())
  self.k.set_length(self.listmap.len())
  self.v.set_length(self.listmap.len())
  # A solution not based on "unflatten" would have been more generic.
  self.k[new_values] = self.k.stub().unflatten(new_keys)

def _raw_update(self, other):
  if type(other) is not Dict:
    raise TypeError("Expected a Dict as parameter.")
  self._raw_update_key(other.keys())

def update(self, other):
  self._raw_update(other)
  self.v[self.listmap[other.keys().flatten()]] = other.values()

def __iadd__(self, other):
  self._raw_update(other)
  self.v[self.listmap[other.keys().flatten()]] += other.values()
  return self

def __isub__(self, other):
  self._raw_update(other)
  self.v[self.listmap[other.keys().flatten()]] -= other.values()
```

```
def __imul__(self, other):
  # This can be optimised. As is, it creates new keys with zero values,
  # then multiplies the zeroes by the values of "other".
  self._raw_update(other)
  self.v[self.listmap[other.keys().flatten()]] *= other.values()

def __itruediv__(self, other):
  # As in __imul__, this can be optimised.
  self._raw_update(other)
  self.v[self.listmap[other.keys().flatten()]] /= other.values()

def __ifloordiv__(self, other):
  # As in __imul__, this can be optimised.
  self._raw_update(other)
  self.v[self.listmap[other.keys().flatten()]] //= other.values()

def len(self):
  return self.keys().len()

def keys(self):
  return self.k

def values(self):
  return self.v

def __getitem__(self, k):
  return self.v[self.listmap[k.flatten()]]

def lookup(self, k, default = None):
  # default must be of the same type as self.v or None
  if default is None:
    default = self.v.stub().set_length(1)
  indices = self.listmap.lookup(k.flatten())
  return fl.mux(indices != -1, self.v[indices], default)
    # This previous line fails if default is not of the same type as self.v

def reduce_sum(self, key, value):
  self._raw_update_key(key)
  return self.v.reduce_sum(self.listmap[key.flatten()], value)

def reduce_isum(self, key, value):
  self._raw_update_key(key)
  self.v.reduce_isum(self.listmap[key.flatten()], value)

def contains(self, k):
  return self.listmap.contains(k.flatten())

def __delitems__(self, k):
  (moved_keys, old_values, new_values) = \
                                  self.listmap.remove_items(k.flatten())
  self.v[new_values] = self.v[old_values]
  self.k[new_values] = self.k[old_values]
  self.v.set_length(self.listmap.len())
```

```
    self.k.set_length(self.listmap.len())

  def discard_items(self, k):
    (moved_keys, old_values, new_values) = \
                                    self.listmap.discard_items(k.flatten())
    self.v[new_values] = self.v[old_values]
    self.k[new_values] = self.k[old_values]
    self.v.set_length(self.listmap.len())
    self.k.set_length(self.listmap.len())


class DictTransmitter(fl.Transmitter):
  def __init__(self, obj):
    super().__init__(obj.context())
    self.k = obj.k.transmitter()
    self.v = obj.v.transmitter()
  def transmit(self, map):
    if type(map) is not dict:
      raise TypeError("Expected dict argument in 'transmit'.")
    self.context().verify_context(map.values())
    if not set(map.keys()).issubset(self.context().scope()):
      raise KeyError("Map keys are out of scope in 'transmit'.")
    for k in map:
      if type(map[k]) is not Dict:
        raise TypeError("Map values incompatible with transmitter.")
      if not map[k].scope().issubset(self.context().scope()):
        raise KeyError("Transmitted values are outside execution scope.")
    # The following two lines fail if items are not of the same type as obj.
    tk = self.k.transmit({n : item.k for n, item in map.items()})
    tv = self.v.transmit({n : item.v for n, item in map.items()})
    out_map = {}
    for n in tk.keys():
      with fl.on(tk[n].scope()):
        out_map[n] = Dict(tk[n], tv[n])
    return out_map
```

8.3 Sparse matrix by vector multiplication

We will now demonstrate how to compute a product, Mv, of a sparse matrix M and a sparse vector v.

Suppose the M is represented by a `Dict` whose keys are `Pair`s of integers and whose values are integers. The pairs represent (x, y) coordinates, and the values are M's values in those coordinates. The rest of M is assumed to be zeroes.

Similarly, v is represented by a `Dict` mapping from integer vector positions to integer vector values, with all other positions assumed to be zero.

We can now compute the product as follows (with the result also being a sparse vector):

```
def matvecmult(m,v):
  rc = v.stub()
  with fl.massop():
    rc.reduce_sum(m.keys().second, m.values() * v[m.keys().first])
  return rc
```

Note that the same code works also if the values in M and v are of types other than integers, as long as multiplying values of the element type of M by values of the element type of v is supported

and yields the same element type as v, and v supports "`.reduce_sum()`".

8.4 ElGamal cryptosystem

We show how to implement the ElGamal cryptosystem using only the primitives afforded by `ftillite`.

For this, we define a new `ElGamalCipher` type, deriving from `Pair`.

```
class ElGamalCipher(Pair):
  def __init__(self, mask, masked_message):
    if type(mask) is not fl.Ed25519ArrayIdentifier or \
        type(masked_message) is not fl.Ed25519ArrayIdentifier:
      raise TypeError("Unexpected argument types constructing ElGamalCipher.")
    super().__init__(mask, masked_message)

  def transmitter(self):
    return ElGamalCipherTransmitter(self)

  def stub(self):
    return ElGamalCipher(self.first.stub(), self.second.stub())

  def sametype(self, other):
    return type(other) is ElGamalCipher and \
             self.first.sametype(other.first) and \
             self.second.sametype(other.second)

  def __eq__(self, other):
    if type(other) is not ElGamalCipher:
        return 0
    return (self.first == other.first) & (self.second == other.second)

  def __ne__(self, other):
    return 1 - (self == other)

  def copy(self):
    return ElGamalCipher(self.first, self.second)

  def promote(self, typecode):
    if self.typecode() != typecode:
      raise TypeError("Cannot promote an ElGamalCipher")
    return (self, False)

  def astype(self, typecode):
    if self.typecode() != typecode:
      raise TypeError("Cannot convert an ElGamalCipher")
    return self.copy()

  def __getitem__(self, key):
    return ElGamalCipher(self.first[key], self.second[key])

  def lookup(self, key, default = None):
    if default is None:
      return ElGamalCipher(self.first.lookup(key), self.second.lookup(key))
    elif self.sametype(default):
```

```
    return ElGamalCipher(self.first.lookup(key, default.first), \
                         self.second.lookup(key, default.second))
  else:
    raise TypeError("Unexpected type for default.")

def reduce_sum(self, key, value):
  if not self.sametype(value):
    raise TypeError("Value is not the same type as LHS.")
  self.first.reduce_sum(key, value.first)
  self.second.reduce_sum(key, value.second)

def reduce_isum(self, key, value):
  if not self.sametype(value):
    raise TypeError("Value is not the same type as LHS.")
  self.first.reduce_isum(key, value.first)
  self.second.reduce_isum(key, value.second)

def cumsum(self):
  self.first.cumsum()
  self.second.cumsum()

def __setitem__(self, key, value):
  if type(value) is not ElGamalCipher:
    raise TypeError("Cannot set an ElGamalCipher to a different type value.")
  self.first[key] = value.first
  self.second[key] = value.second

def broadcast_value(self, length):
  (first, is_copy1) = self.first.broadcast_value(length)
  (second, is_copy2) = self.second.broadcast_value(length)
  if not (is_copy1 or is_copy2):
    return (self, False)
  return (ElGamalCipher(first, second), True)

def __mux__(self, conditional, iffalse):
  if type(iffalse) is not ElGamalCipher:
    raise TypeError("Both sides of mux must have same type.")
  return ElGamalCipher(fl.mux(conditional, self.first, iffalse.first), \
                       fl.mux(conditional, self.second, iffalse.second))

# There is no need to implement __rmux__, because nothing can be promoted to
# an ElGamalCipher.

def decrypt(self, priv_key):
  fc = fl.get_context([self, priv_key])
  # The following checks are not needed. Without them, the same errors will
  # cause the same exceptions to be raised, only a few lines further down.
  if not fc.scope().issubset(self.scope()) \
       or not fc.scope().issubset(priv_key.scope()):
    raise KeyError("Execution scope exceeds data scope.")
  fl.verify(priv_key.len() == 1)
  # This check is not needed, either.
  assert(type(priv_key) is fl.Ed25519IntArrayIdentifier)
  with fl.massop():
```

```
    rc = self.second - self.first * priv_key
  return rc

def __pos__(self):
  return self  # No real need to return a copy here.

def __neg__(self):
  return ElGamalCipher(-self.first, -self.second)

def __add__(self, other):
  if type(other) is not ElGamalCipher:
    return NotImplemented
  fc = self.context()
  length = fc.calc_broadcast_length([self, other])
  (other, is_copy) = other.broadcast_value(length)
  (self2, is_copy) = self.broadcast_value(length)
  return ElGamalCipher(self2.first + other.first, \
                       self2.second + other.second)

def __iadd__(self, other):
  if type(other) is not ElGamalCipher:
    raise TypeError("Incompatible types for addition.")
  (other, is_copy) = other.broadcast_value(self.len())
  self.first += other.first
  self.second += other.second
  return self

def __sub__(self, other):
  if type(other) is not ElGamalCipher:
    return NotImplemented
  fc = self.context()
  length = fc.calc_broadcast_length([self, other])
  (other, is_copy) = other.broadcast_value(length)
  (self2, is_copy) = self.broadcast_value(length)
  return ElGamalCipher(self2.first - other.first, \
                       self2.second - other.second)

def __isub__(self, other):
  if type(other) is not ElGamalCipher:
    raise TypeError("Incompatible types for subtraction.")
  (other, is_copy) = other.broadcast_value(self.len())
  self.first -= other.first
  self.second -= other.second

def __mul__(self, other):
  fc = self.context()
  fc.promote(other)
  if not isinstance(other, \
                    (fl.IntArrayIdentifier, fl.Ed25519IntArrayIdentifier)):
    return NotImplemented
  length = fc.calc_broadcast_length([self, other])
  (other, is_copy) = other.broadcast_value(length)
  (self2, is_copy) = self.broadcast_value(length)
  return ElGamalCipher(self2.first * other, \
```

```
                          self2.second * other)

  def __imul__(self, other):
    fc = self.context()
    fc.promote(other)
    if not isinstance(other, \
                      (fl.IntArrayIdentifier, fl.Ed25519IntArrayIdentifier)):
      raise TypeError("Incompatible types for multiplication.")
    (other, is_copy) = other.broadcast_value(self.len())
    self.first *= other
    self.second *= other

  def __rmul__(self, other):
    return self * other


class ElGamalCipherTransmitter(PairTransmitter):
  def __init__(self, obj):
    super().__init__(obj)
  def transmit(self, map):
    if type(map) is not dict:
      raise TypeError("Expected dict argument in 'transmit'.")
    self.context().verify_context(map.values())
    if not set(map.keys()).issubset(self.context().scope()):
      raise KeyError("Map keys are out of scope in 'transmit'.")
    for k in map:
      if type(map[k]) is not ElGamalCipher:
        raise TypeError("Map values incompatible with transmitter.")
      if not map[k].scope().issubset(self.context().scope()):
        raise KeyError("Transmitted values are outside execution scope.")
    t1 = self.first.transmit({n : item.first for n, item in map.items()})
    t2 = self.second.transmit({n : item.second for n, item in map.items()})
    out_map = {}
    for n in t1.keys():
      with fl.on(t1[n].scope()):
        out_map[n] = ElGamalCipher(t1[n], t2[n])
    return out_map



def elgamal_encrypt(plaintext, pub_key):
  # For simplicity, this example does not feature a zero stockpile.
  #
  assert(type(pub_key) is fl.Ed25519ArrayIdentifier)
  fl.verify(pub_key.len() == 1) # This also verifies the scope of pub_key.
  # note that len() of an fc.array is a singleton fc.array, which may have a
  # different value in each node.
  fc = fl.get_context([plaintext, pub_key])
  (plaintext, is_copy) = fc.promote(plaintext)
    # The above line also verifies the scope of plaintext.
  with fl.massop(): # This context defines a single mass operation.
    nonce = fc.randomarray('I', plaintext.len(), False)
    rc = ElGamalCipher(nonce.astype('E'), \
                       nonce * pub_key + plaintext.astype('E'))
```

```
    del nonce
  return rc

def elgamal_refresh(ciphertext, pub_key):
  fc = fl.get_context([ciphertext, pub_key])
  if type(ciphertext) is not ElGamalCipher:
    raise TypeError("Expected ElGamalCipher as ciphertext in refresh.")
  zero = fc.array('i', ciphertext.len())
  nonce = elgamal_encrypt(zero, pub_key)
  ciphertext += nonce

def elgamal_sanitise(ciphertext):
  if type(ciphertext) is not ElGamalCipher:
    raise TypeError("Expected ElGamalCipher in elgamal_sanitise.")
  mask = ciphertext.context().randomarray('I', ciphertext.len(), True)
  ciphertext *= mask

def elgamal_keygen(fc):
  priv_key = fc.randomarray('I', 1)
  pub_key = priv_key.astype('E')
  return (priv_key, pub_key)
```

To demonstrate the system in action, let's prepare some data.

```
with fl.on(fc.CoordinatorID): # On the coordinator node...

  # Creating an ElGamal key pair.
  (priv_key, local_pub_key) = elgamal_keygen(fc)

# Back on all nodes: distributing the public key across all nodes.
pub_key = fl.transmit({i : local_pub_key for i in fc.scope()})[fc.CoordinatorID]

# On the peer nodes...
peer_nodes = fc.scope().difference(fc.CoordinatorID)
with fl.on(peer_nodes):

  # Reading some data. ("query" is a string with an appropriate SQL query.)
  my_data = fc.auxdb_read(query, 'i')

# Back on all nodes again.
```

Now, here's code that encrypts the data in the peer nodes, sends it over to the coordinator node, and decrypts it there. (This is not something that one would normally do in FinTracer. It's just given here to show an example of `ftillite` in action.)

```
with fl.on(peer_nodes):
  encrypted_data = elgamal_encrypt(my_data, pub_key)

local_encrypted_data = fl.transmit({fc.CoordinatorID : encrypted_data})

with fl.on(fc.CoordinatorID):
  local_decrypted_data = {n : data.decrypt(priv_key) \
                              for n, data in local_encrypted_data.items()}
```

8.5 FinTracer step

The code for matrix multiplication is almost identical to what we need for performing a FinTracer step.

Suppose that we define accounts by means of pairs of account numbers and account BSB, both of which are represented by integers.

```
accounts = Pair(fc.array('i'), fc.array('i'))
```

And suppose further that transactions are stored merely as pairs of accounts, first the "from" account, and then the "to" account.

```
transactions = Pair(accounts, accounts)
```

Let us populate a transaction graph (which we'll assume to be valid, to keep the example short), by a SQL query.

```
with fl.on(peer_nodes):
  transactions.auxdb_read(query)
```

A FinTracer tag is a mapping from accounts to ElGamal ciphertexts. Let us first have AUSTRAC read a set of accounts via a SQL query and populate a Dict mapping from those to ciphertexts encrypting "1" values.

```
with fl.on(fc.CoordinatorID):
  accounts.auxdb_read(query)
  values = fc.array('i', accounts.len(), 1)
  ciphertext = elgamal_encrypt(values, pub_key)
  tag = Dict(accounts, ciphertext)
```

Next, let us build a function that can distribute the values of a tag, and use it to bring each portion of the tag to its rightful owner.

We refresh all tags before sending them, for which reason we need the public key, but the code demonstrates that we can avoid refreshing tags that stay within a node.

First we devise a mapping that will tell us which BSB goes with which node. We'll actually use the BSB data on the nodes to create the mapping, because this mapping may change over time.

```
def get_branch2node(fc):
  branch2node_first = fc.array('i', 1000000, -1)
  branch2node_last = fc.array('i', 1000000, -1)
  all_accounts = Pair(fc.array('i'), fc.array('i'))
  peer_nodes = fc.scope().difference(fc.CoordinatorID)
  with fl.on(peer_nodes):
    bsbs = fc.array('i')
    bsbs.auxdb_read("SELECT DISTINCT bsb FROM accounts;")
  bsbs = fl.transmit({i : bsbs for i in fc.scope()})
  k = bsbs.keys()
  for i in k:
    branch2node_last[bsbs[i]] = i.num()
  for i in reversed(k):
    branch2node_first[bsbs[i]] = i.num()
  fl.verify(branch2node_first == branch2node_last)
  return branch2node_first

branch2node = get_branch2node(fc)
```

```
def distribute_tag(tag, pub_key):
  out_scope = tag.context().scope().difference([tag.context().CoordinatorID])
  map = {}
  with fl.on(tag.scope()):
    for n in out_scope:
      # This can be done more efficiently using "discard_items".
      ind = (branch2node[tag.keys().first] == n.num()).index()
      map[n] = Dict(tag.keys()[ind], tag.values()[ind])
  # The next loop performs refreshing, prior to transmitting.
  for n in out_scope:
    with fl.on(tag.scope().difference(set([n]))):
      elgamal_refresh(map[n].values(), pub_key)
  return fl.transmit(map)


dist_tag = distribute_tag(tag, pub_key)[fc.CoordinatorID]
```

Now, let us take this distributed tag, and perform on it one FinTracer step.

```
def fintracer_step(tag, txs, pub_key):
  fc = fl.get_context([tag, txs, pub_key])
  with fl.on(tag.scope()):
    zero = fc.array('i', 1)
    nonce = elgamal_encrypt(zero, pub_key)
    next_tag = tag.stub()
    with fl.massop():
      next_tag.reduce_sum(txs.second, tag.lookup(txs.first, nonce))
  map = distribute_tag(next_tag, pub_key)
  scope = set()
  for n in map:
    scope.update(map[n].scope())
  with fl.on(scope):
    # We need to create here a new empty tag, but can't use ".stub()" because
    # it may have a different scope than any existing tag.
    # In a real implementation, we would have had a "tag" type, and all this
    # would have happened automatically.
    accounts_stub = Pair(fc.array('i'), fc.array('i'))
    cipher_stub = ElGamalCipher(fc.array('E'), fc.array('E'))
    rc = Dict(accounts_stub, cipher_stub)
    for n in map:
      with fl.on(map[n].scope()):
        rc += map[n]
  return rc


dist_tag = fintracer_step(dist_tag, transactions, pub_key)
```

A Possible implementations

The present document does not advocate any particular implementation method. However, in this appendix we provide, purely for convenience and disambiguation, reference pure-Python implementations of some of the objects described.

A.1 ManagedObject

The following is a possible implementation of `ManagedObject`:

```
class ManagedObject:
  def __init__(self, fc):
    self._context = fc
  def context(self):
    return self._context
```

A.2 NodeSet

The following is a fairly minimalist possible implementation of `NodeSet`. The general idea is that in any operation that alters the `NodeSet`, we make sure that the resulting set is valid: all set elements remain nodes that all belong to the original `FTILContext` object.

Not included in our implementation are methods like "`__and__`" that, without overwriting them, will not fail on an input that will cause, e.g., "`__or__`" to fail, but will, on the other hand, never cause a `NodeSet` with elements of an inconsistent context to be created.

```
class NodeSet(set, ManagedObject):
    def __init__(self, val):
        if type(val) is Node:
            set.__init__(self, [val])
            ManagedObject.__init__(self, val.context())
        elif isinstance(val, FTILContext):
            set.__init__(self)
            ManagedObject.__init__(self, val)
        else:
            if isinstance(val, NodeSet):
                fc = val.context()
            else:
                fc = get_context(val)
            for e in val:
                if type(e) is not Node:
                    raise TypeError("Elements sent to NodeSet are not nodes.")
            set.__init__(self, val)
            ManagedObject.__init__(self, fc)

    def __ior__(self, other):
        if isinstance(other, FTILContext):
            raise TypeError("Unexpected parameter type: FTILContext.")
        other = NodeSet(other)
        if self.context() != other.context():
            raise ValueError("NodeSets contexts mismatch.")
        set.__ior__(self, other)
        self = NodeSet(self)

    def __ixor__(self, other):
        if isinstance(other, FTILContext):
            raise TypeError("Unexpected parameter type: FTILContext.")
        other = NodeSet(other)
        if self.context() != other.context():
            raise ValueError("NodeSets contexts mismatch.")
        set.__ixor__(self, other)
        self = NodeSet(self)
```

```
def __or__(self, other):
    if isinstance(other, FTILContext):
        raise TypeError("Unexpected parameter type: FTILContext.")
    other = NodeSet(other)
    if self.context() != other.context():
        raise ValueError("NodeSets contexts mismatch.")
    return NodeSet(set.__or__(self, other))

def __ror__(self, other):
    if isinstance(other, FTILContext):
        raise TypeError("Unexpected parameter type: FTILContext.")
    other = NodeSet(other)
    if self.context() != other.context():
        raise ValueError("NodeSets contexts mismatch.")
    return NodeSet(set.__ror__(self, other))

def __rxor__(self, other):
    if isinstance(other, FTILContext):
        raise TypeError("Unexpected parameter type: FTILContext.")
    other = NodeSet(other)
    if self.context() != other.context():
        raise ValueError("NodeSets contexts mismatch.")
    return NodeSet(set.__rxor__(self, other))

def __xor__(self, other):
    if isinstance(other, FTILContext):
        raise TypeError("Unexpected parameter type: FTILContext.")
    other = NodeSet(other)
    if self.context() != other.context():
        raise ValueError("NodeSets contexts mismatch.")
    return NodeSet(set.__xor__(self, other))

def add(self, other):
    if type(other) is not Node:
        raise TypeError("Unexpected parameter type. Expected Node.")
    if self.context() != other.context():
        raise ValueError("NodeSet and parameter contexts mismatch.")
    set.add(self, other)
    self = NodeSet(self)

def difference(self, other):
    if isinstance(other, FTILContext):
        raise TypeError("Unexpected parameter type: FTILContext.")
    other = NodeSet(other)
    if self.context() != other.context():
        raise ValueError("NodeSets contexts mismatch.")
    return NodeSet(set.difference(self, other))

def symmetric_difference(self, other):
    if isinstance(other, FTILContext):
        raise TypeError("Unexpected parameter type: FTILContext.")
    other = NodeSet(other)
    if self.context() != other.context():
```

```
            raise ValueError("NodeSets contexts mismatch.")
        return NodeSet(set.symmetric_difference(self, other))

    def symmetric_difference_update(self, other):
        if isinstance(other, FTILContext):
            raise TypeError("Unexpected parameter type: FTILContext.")
        other = NodeSet(other)
        if self.context() != other.context():
            raise ValueError("NodeSets contexts mismatch.")
        set.symmetric_difference_update(self, other)
        self = NodeSet(self)

    def union(self, other):
        if isinstance(other, FTILContext):
            raise TypeError("Unexpected parameter type: FTILContext.")
        other = NodeSet(other)
        if self.context() != other.context():
            raise ValueError("NodeSets contexts mismatch.")
        return NodeSet(set.union(self, other))

    def update(self, other):
        if isinstance(other, FTILContext):
            raise TypeError("Unexpected parameter type: FTILContext.")
        other = NodeSet(other)
        if self.context() != other.context():
            raise ValueError("NodeSets contexts mismatch.")
        set.update(self, other)
        self = NodeSet(self)

    def copy(self):
        return NodeSet(set.copy(self))
```

A.3 `on`

Suppose that internally to `ftillite`, not externalised to the user, are the following two commands:

`fc._push_scope(new_scope)`: Adds a new scope to the top of the execution scope stack. This now becomes the new active scope.

`fc._pop_scope()`: Removes the top of the execution scope stack and sets the current scope to its previous value.

Then the requested "`with fl.on`" interface is semantically equivalent to `fl` having the following class definition:

```
class on:
  def __init__(self, new_scope):
    self.new_scope = NodeSet(new_scope)
    if len(self.new_scope) == 0:
      raise ValueError("Scope cannot be empty.")
    self.context = self.new_scope.context()
    if not self.new_scope.issubset(self.context.scope()):
      raise ValueError("New scope must be a subset of existing scope.")
  def __enter__(self):
      self.context._push_scope(self.new_scope)
```

```
      return self.context.scope()
  def __exit__(self, exc_type, exc_value, traceback):
    self.context._pop_scope()
```

Note A.1. In this implementation, `_push_scope` can assume that its input is a valid, nonempty `NodeSet`, so no further checking of the input is needed.

A.4 `Identifier`

A possible implementation of `Identifier` is as follows.

```
import re

def typecode_list(tc):
    return re.split("([ifIE]|b[1-9][0-9]*)\s*", tc)[1::2]

class Identifier(ManagedObject):
    def __init__(self, fc, **kwargs):
        super().__init__(fc)
        if not isinstance(self.context().scope(), NodeSet):
            raise ValueError("scope is not of type NodeSet")
        self._scope = self.context().scope()

    def scope(self):
        if not isinstance(self._scope, NodeSet):
            raise ValueError("scope is not of type NodeSet")
        return self._scope

    def width(self):
        return len(typecode_list(self.typecode()))

    def auxdb_read(self, query):
        flat = self.context().auxdb_read(query, self.typecode())
        if type(flat) is list:
            self.unflatten(flat)
        else:
            self.unflatten([flat])

    def typecode(self):
        return "".join([x.typecode() for x in self.flatten()])

    def __getstate__(self):
        state = {}
        state["_scope"] = [x.num() for x in self._scope]
        return state

    def __setstate__(self, state):
        # Python requires us to have a global "backend" object for this.
        self._context = self._backend.context()
        if self._context == None:
            raise RuntimeError("Cannot load when not in a session.")
        self._scope = [self.context().Nodes[x] for x in state["_scope"]]
```

A.5 `ArrayIdentifier`

The following is a possible Python implementation for `ArrayIdentifier`.

To implement `broadcast_value` we assume the existence of a function "`fl._equal_int`" that accepts two singleton `IntArrayIdentifier`s and returns as a Python Boolean whether they are equal in every node.

Note A.2. The "`fl._equal_int`" functionality is required because `ftillite`'s handling of "handles" is much simpler than that of FTIL. It is not, in general, a safe function to have as it can retrieve private information from remote nodes.

The "`fl._equal_int`" is assumed to perform no conversions, promotions or broadcasting of any kind. Its expectations regarding its parameter types are strict. In particular, if either argument sent to it is not a singleton, it should raise a `ValueError` exception.

Similarly, we assume that `FTILContext` has a method "`_raw_broadcast_value(var, len)`" that performs, essentially, a "`var.set_length(len)`", but without triggering any attempts to promote or convert the "`len`" parameter.

```
class ArrayIdentifier(Identifier):
    def __init__(self, fc, **kwargs):
        super().__init__(fc=fc, **kwargs)

    def width(self):
        return 1

    def transmitter(self):
        return BuiltinTransmitter(self)

    def stub(self):
        return self.context().array(self.typecode())

    def flatten(self):
        return [self]

    def copy(self):
        return self[:]

    def promote(self, typecode):
        if self.typecode() != typecode:
            raise TypeError("Cannot promote to the requested type.")
        return (self, False)

    def empty(self):
        self.set_length(0)

    def broadcast_value(self, length):
        if type(length) is int:
            length = self.context().array("i", 1, length)
        if type(length) is not IntArrayIdentifier:
            raise TypeError("Length parameter must be IntArrayIdentifier.")
        self.context().verify_context([length])
        lenlen = length.len()
        if _equal_int(self.len(), length):
            return (self, False)
        elif _equal_int(self.len(), lenlen):
```

```
        copy = self.copy()
        fc._raw_broadcast_value(copy, length)
        return (copy, True)
    else:
        raise ValueError("Value not eligible for broadcasting.")
```

B Changelog

B.1 May 5, 2022

First official release of the `ftillite` document.

B.2 June 8, 2022

- The method "`.ed_affine()`" was replaced by the method "`.ed_folded()`" which returns the folded representation.
- The method "`.sha3_256_keygen()`" was removed. To create a keyed hash, concatenate a key to the data-to-hash before hashing it.
- A new method "`.concat()`" was introduced to conveniently concatenate two `Bytearray` arrays (presumably more efficiently than through projection and bit-operations alone).
- How to generate a random uniform float is now specified explicitly.
- The `FtilliteBuiltin` class is mentioned.
- The `ArrayTransmitter` and `ListMapTransmitter` classes have been unified into a new `BuiltinTransmitter`.
- The function "`fl.assert`" had its name changed to "`fl.verify`" because "`assert`" is a Python keyword.
- Instead of "`len(x)`" returning, in `IntArrayIdentifier` form, the length of `x` across all nodes, this is now done using "`x.len()`". The construct "`len(x)`" still exists, but returns the length of `x` in the coordinator node only (if it is defined there), and its return value is a Python native `int`. This is due to Python's use of an object's `__len__` property when converting it to a list.

B.3 June 30, 2022

- Added a new "`auxdb_write`" function.
- Removed the convenience function "`fc.branch2node(branches)`". The mapping seems too volatile to be set in software. The `ftillite` user will, instead, need to create a function that generates such a mapping from the data existing at the RE nodes, noting that BSB association is not private information, so there is no problem with sending it from the peer nodes to the coordinator node.

B.4 July 11, 2022

- Section 3.3 was added, to clarify how to handle exceptions.
- Methods named "`__contains__`" changed their names to "`contains`", to account for the fact that Python does not allow the magic method "`__contains__`" to be overwritten in the way intended by `ftillite`.

B.5 July 21, 2022

- Now supporting both "`ed_affine`" and "`ed_folded`".
- When converting from affine to projective representation, the name of the new method is "`ed_affine_project`".
- Correspondingly, "`ed_project`" changed its name to "`ed_folded_project`".

B.6 July 28, 2022

- The method "`unflatten`" now unamiguously returns the modified object.

B.7 August 4, 2022

- The `listmap` method "`todict`" was introduced, replacing any conversion-to-dictionary operator.
- All code listings in Section 8 and Appendix A have been updated to real working code.

B.8 September 28, 2022

- The specs of the `calc_broadcast_length` method changed to properly handle the case where some lengths are zero, some are one, and there are no other lengths. The specs originally stipulated that the return value in this case will be one. This has now been corrected to zero.

References

[1] Brand, M. (2022), *The Complete, Authorised and Definitive FinTracer Compendium (Volume 1, Part 1, 3rd Revision [a.k.a. "Purgles" edition]): A work in progress.* (Available as: `FT_combined.pdf`)