Swift vs OPM Collections

Ben Clifford, benc@hawaga.org.uk

April 7, 2010

This note contrasts OPM collections as proposed in [OC] with similar collections present in Swift. First, there is a brief overview of collections in OPM and Swift. Secondly each part of the OPM proposal (section 4 of [OC]) is compared to Swift. Thirdly, some issues to do with arrays, mappers and the OPM model of time are discussed.

This note makes no attempt to be an introduction to OPM, OPM collections or Swift, and other sources should be consulted for such material.

1 Swift collections

Swift refers to all data through a *DSHandle*, which is close (though not identical) to an OPM artifact. Basic data types are represented as DSHandles that refer to files on disk or to in-memory data such as integers.

DSHandles can represent collections of other DSHandles. Two forms are provided for this: structures (what these are actually called has always been a little ambiguous) and arrays. These are analogous to structs and arrays in C: a struct contains a predefined number of DSHandles referenced by name, and the type of each of those named members is known statically (i.e. at compile time); arrays contain an arbitrary number of DSHandles referenced by an integer index, and the type of each element is the same. The syntax used to define and refer to structures and arrays is reminiscent of C: `data.x` for structures, and `a[5]` for arrays.

Swift collections are immutable: a particular member has only a single value, and so a reference to a particular member is unambiguous. However, collections are constructed over time.

2 OPM collections

The OPM definition of collections has a different feel, as it is naturally centered around provenance. The following sections will address each part of the proposal with respect to Swift collections.

3 Collections and artifacts (4.1)

Broadly speaking, a Swift DSHandle maps to an OPM artifact, whether for collection artifacts or for contained artifacts. There are some differences, which are discussed in the following sections.

4 Structural level (4.2)

[OC] proposes a *Contained* relation between two artifacts, indicating that one is contained within another. This maps relatively cleanly to Swift's in-memory model of DSHandles containing other DSHandles, whether in structs or arrays, although some issues with aliasing arise, as discussed in the next section.

Swift collections may be hierarchical. [OC] does not specify if the Contained relation holds only one level deep or to all elements contained in a collection.

The Contained relation as described in [OC] does not have any annotations. In Swift, there is always an explicit path (a structure member name, or an array index, or for deep nesting a sequence of those) from a collection to its members, and that path might be placed an annotation on the contained relation.

5 Constructor and accessor level (4.3)

[OC] proposes two types of processes: *constructors* and *accessors*. Constructors derive collection artifacts from artifacts representing members of those collections. Conversely, accessors derive artifacts representing (some or all) members from the corresponding collection artifact.

Note that an artifact retrieved from a collection is never the same artifact that went into the constructor for that collection. This reflects the provenance oriented view that, although the underlying data may be the same, there may be additional provenance (for example, the choice of that particular data). This is discussed further in section 8

In the present Swift implementation, a DSHandle returned from a collection accessor is the same DSHandle used to construct that accessor. Whilst this is sufficient for providing access to the relevant data at runtime, it is unsufficient for describing the provenance of collection accesses. A suggested modification to Swift is to create new DSHandles on collection access, rather than returning existing DSHandles.

6 Communication level (4.4)

[OC] proposes a relation *WasCopyOf* between artifacts that indicates that one artifact is a copy of another artifact (perhaps at a different location). At the SwiftScript level, data transfer is never explicitly expressed, and happens at a lower level than the Swift provenance work has attempted to document. Thus the relations between artifacts described in this section are irrelevant to Swift.

7 Operation level (4.5)

[OC] proposes a relation *WasMappedFrom* between two collections, indicating that for each element in the 'destination' collection has been derived by some process applied to a corresponding element of the 'source' collection.

This relation fits in extremely well with the spirit of Swift, where one of the basic use cases is to apply the same operation to a very large number of input data files. However, it does not correspond well to the semantics of the SwiftScript language, and it is not straightforward to express WasMappedFrom given a SwiftScript code fragment implementing that map. This is a weakness in SwiftScript. For other reasons, I've suggested implementing map-like control structures, and *WasMappedFrom* is in line with those.

8 Array References

When an artifact is placed into a collection, or accessed from that collection, that access can be annotated with the position in the collection. In the case of a structure, this annotation is the member name. In the case of an array, this annotation is the array index.

These two annotations differ superficially in their type - one is a SwiftScript identifier, a string; the other is an integer. However there is a more fundamental difference: structure references are static, in the sense that the identifier is hard-coded into a SwiftScript program: the SwiftScript program fragment `s.left` refers to the `left` element of the collection s, and this can be observed by inspection of the source code. But array references can be dynamically constructed at runtime. It is not possible by source code inspection to determine which element is being accessed by `a[i]`. Instead, `i` is itself a DSHandle (or artifact) with its own provenance, and that provenance is relevant to any further process which takes `a[i]` as input.

How, then, does OPM represent the provenance of this index when accessing an array? Two approaches spring to mind: firstly, that annotations in general can have provenance (for example, by annotations being artifacts in their own right); and secondly that a different kind of accessor, which is uses not only a collection artifact (as for the present [OC]), but also another artifact representing the index. It is not clear to me what the (dis)advantages of each approach are, from the perspective of accessing arrays.

More confusingly, though, is the case of array construction. In Swift, this happens piecewise:

```
a[i]=3; a[j]=10;
```

In this situation, I think I favour the first of the above approaches: a single constructor process constructs the array, with annotations on each 'uses' relation indicating the runtime values of `i` and `j`, and with those annotations having provenance.

In practice with SwiftScript, its very common to assign arrays using a dynamically constructed index (where that index is coming from some enclosing foreach loop), but the higher level concept being expressed is often simpler - for example, a map as described in section 7.

9 Mappers and filenames

In Swift, mappers define both where data will be stored, and in the case of input collections what data will be in those collections. Every DSHandle that represents out of core data has a filename (or more generally a URI) which describes where that data will be stored.

Mappers are supplied with parameters, which may be dynamically computed. Thus, like array references, what starts as a simple annotation now has provenance to be recorded.

For input collections, mappers look very like constructors: they use a set of existing artifacts to produce a collection artifact. But what about collections that are generated by a Swift program? How does the mapper fit into the provenance graph here? Its not at all clear to me...

10 Time

Collections in Swift are constructed over time but result in a single collection artifact representing the final state of that collection.

A partially constructed collection can never be observed by a SwiftScript program - when Swift attempts to evaluate an expression referring to part of a collection that does not exist, evaluation is deferring until that part does exist. That is fundamental to Swift's execution model.

Parts of a collection may be used before the construction of an entire collection is complete. That again is fairly fundamental.

The collection artifact is integral here:

```
a[i]=10;
o=f(a[j]);
```

If `i==j` then the above two statements can both run without the collection `a` being fully constructed; but there is an array accessor that uses `a`

How does this interact with the OPM time model? OPM 1.1 makes places certain requirements on the time annotations for processes and artifacts that might not map well to the Swift provenance model.

A collection artifact is constructed by a constructor process. In Swift, such a process is implicit and not related to a single language statement, (although in a tighter language specification it might be, but this situation still exists).

Consider the creation of a 2-element collection, where the first element is used before the second element is created: (forgive the formatting - graphviz would do a much better job)

```
i1 and i2 are artifacts.
C is the constructor
c is the collection
A is the accessor
a1 is the accessed version of i1

i1 <- used <- C (t1)
i2 <- used <- C (t2)
C <- generatedBy <- c (t3)
c <- used <- A (t4)
A <- generatedBy <- a1 (t5)
```

OPMv1.1 specifies the constraint that an artifact must exist before being used, that `t3<t4`, and so that the constructor process must have generated the collection before the accessor process can use it. This suggests that in Swift, the collection should have a generation time of approximately when we place the first object in it.

In our example, i2 is generated (by circumstances of fate rather than explicit instruction in a SwiftScript program) after a1 has been used.

So now, the constructor uses i2 after it has generated c.

This matches the rules of OPM times. But it feels like a violation of the spirit.

Please discuss...?

11 Exposing the dynamic behaviour of collection construction

The previous sections describe one particular representation of collections in Swift that is aligned with the immutable-variable view of Swift's data model. Another representation, not pursued but described briefly here, tracks the state of collections as time progresses.

In this alternate representation, the state of a collection at any point would be represented by an artifact. As processes which generate members complete, a new artifact representing the updated state of the collection comes into existence, with relations showing its derivation from the previous state and from the new member artifacts.

The problem with times discussed in section 10 does not appear in this model, as an accessor only uses an artifact that already exists, representing enough of the array to supply the requested member artifact.

This model more accurately reflects the internal state of the Swift runtime over time, and so may be desirable in some cases. However, it less accurately reflects the structure of SwiftScript programs, and so (I think) is likely to be less easily understood and less useful to an application programmer who may wish to reason in terms of their application data flow.

12 Summary of OPM suggestions

Section 4 asks if the *contains* relation is transitive over multiple levels of containment. This is not explicit in the text of [OC]. That section also asks if there is a standard annotation for indicating the name of a member of a collection (an array index or structure member name)

Section 8 asks how provenance of array indices is to be stored, suggesting either that annotations are artifacts in their own right, or that a different style of accessor be defined, using an artifact rather than an annotation to express array index.

Section 10 expresses concern over the violation of the spirit of the OPM time section when modelling the times of Swift collection construction and access.

13 Summary of Swift suggestions

Section 5 suggests that Swift should produce new DSHandles to represent the aliases created by structure or array access in order to better record the provenance of those accesses independently from the accessed data.

Section 7 suggests that an explicit map operator may be a more provenance-friendly construct for iterating over a collection. Section 8 reiterates that.

References

[OC] http://mailman.ecs.soton.ac.uk/pipermail/provenance-challenge-ipaw-info/attachments/20090605/85b3e182/attac
0001.pdf