Referential Integrity with Scala Types

Patrick Prémont
BoldRadius Solutions, Canada
patrick.premont@boldradius.com

Abstract

Referential integrity constraints are critical elements of relational data models, and have found widespread use in industry. However, their benefits in terms of data integrity do not fully extend to program correctness. Constraint violations are identified at run-time and must then be handled appropriately by programs. We show how Scala can be used to build data models and programs where referential integrity is enforced at compile-time. Scala's type system, with its variance annotations and path-dependent types, is especially suited to express these constraints and proofs in a natural manner. We also explore potential compiler improvements that could enhance support for type-checked referential integrity.

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features—Data types and structures

Keywords Referential integrity, Dependent types, Variance, Data structures, Scala

1. Introduction

Software developers using typed functional languages like Scala have generally moved away from using null references. They explicitly document which references may be invalid by the use of the Option type. This discipline allows the compiler to verify that checks are present whenever they are needed, which can lead to increased software reliability.

In many cases, empty Option instances have a meaning within the normal operation of a program. In other cases however, those empty Option should never occur based on the intended invariants of the program. Their occurrence at run-time is symptomatic of an error within the program. Distinguishing between these two scenarios can be difficult, and can lead the developer to apply inappropriate recovery measures in the presence of empty Option instances. Fortunately, it is often possible to refactor programs to eliminate occurrences of Option that represent errors. This is often achieved by encoding more invariants in program types.

We present a technique to encode referential integrity constrains [2] in the Scala type system. Referential integrity is concerned with ensuring that references within a table or a data structure designate a valid entry and are not dangling references.

The value of referential integrity has long been recognized in the context of database systems. We believe it to also be valuable to in-memory data structures. In spite of database integrity checks, program errors outside of the database may very well have adverse consequences on persisted data.

Our proposal can increase application reliability significantly by eliminating many occurrences of Option instances, and the associated checks and recoveries. While these empty Option instances appear as a result of data structure lookups, they are in fact symptoms of errors that occur earlier in the program: when data structures are modified in a way that breaks referential integrity. The types we propose will identify those errors during compilation.

2. Types for Referential Integrity

2.1 Current Practice

Programs typically contain data structures whose elements may be looked up by a corresponding key. In Scala, the Map trait abstracts over such data structures. Let us consider only its insertion and lookup methods in relation to referential integrity:

Listing 1. Insertion and lookup in the Map trait

```
trait Map[A, +B] {
  def +[B1 >: B](kv: (A, B1)): Map[A, B1]
  def get(key: A): Option[B]
}
```

The insertion method + yields a Map with the same key type A. As elements are added to the data structure, we keep using the same key type for lookups with get. This constraint forces developers to use a key type that contains

SCALA'15, June 13, 2015, Portland, OR, USA
© 2015 ACM. 978-1-4503-3626-0/15/06...$15.00
http://dx.doi.org/10.1145/2774975.2774979

a large number of values: enough values to accommodate all elements that could potentially be added to the map. A common choice is to use integers.

Using the same key type across unrelated Map instances makes it likely that keys of one map will inadvertently be used to lookup elements of another. A good practice that limits such errors is to create distinct key types for the various maps that have no need to share a key type:

Listing 2. Distinct key types

```
case class CustomerId(value: Int) extends AnyVal
case class AccountId(value: Int) extends AnyVal
val customers: Map[CustomerId, Customer]
val accounts: Map[AccountId, Account]
```

This practice is simple and provides significant safety benefits. However there remains the potential for confusion between the keys of multiple maps. Consider an extension of the example above, where a program comprises a variable number of services, each of which holds a map from the AccountId type:

Listing 3. Remaining ambiguity with distinct key types

```
case class ServiceId(value: Int) extends AnyVal
case class Service(accounts: Map[AccountId, Account])
val services: Map[ServiceId, Service]
```

Here multiple maps of type Map[AccountId, Account] may exist at run-time. The types do not constrain an AccountId to be used for lookups only in maps where it was inserted. Achieving this segregation require a more elaborate representation:

Listing 4. Flexible Key Segregation

```
class AccountIdType {
  case class AccountId(value: Int)
}
trait Service {
  val accountIdType: AccountIdType
  val accounts: Map[accountIdType.AccountId, Account]
}
```

Here we have deployed more elaborate tools, inner classes and path-dependent types, to further reduce the possibility that lookups will be performed on the wrong maps.

The above measures can prevent many map lookups that would have returned None, but they do not achieve the enforcement of referential integrity. As noted earlier, maps must preserve the same key type when elements are added or removed. Lookups for a key that has been removed, or has not yet been inserted, would type-check but would fail to yield a value.

2.2 Lookup as a Total Function

We now show a data structure that is similar to a map but where lookups are guaranteed to yield a value. Here we will refer to keys as *identifiers* since each key reliably refers to a particular value.

The type used for identifiers needs to admit only values that refer to elements present in a particular data structure. This requires an immutable data structure, or at least one in which the set of contained elements may not change.

We propose an immutable *total map* data structure where the insertion and lookup methods are as follows:

Listing 5. Insertion and lookup in a total map

```
sealed trait Total[+V] {
  type Id
  def insert[V2 >: V](value: V2): Extension[Id, V2]
  def apply(id: Id): V
}
trait Extension[K, V] {
  val total: Total[V] {type Id >: K}
  val newId: total.Id
}
```

The essential difference with the Map trait is that the insertion method produces total maps with distinct Id types. For any total map t, each value of type t.Id refers to an existing element of t. Therefore the lookup operation, named apply here, always returns a valid element of type V.

A notable change here is that the type parameter A has been replaced by a type member Id. This avoids extensive reliance on existential types by keeping the output Id types within an object.

We also let the data structure allocate the identifiers during insertion; we do not pass them in. The insert method packages the newly allocated identifier with the resulting Total in an Extension object. A tuple is not sufficient here since we want the identifier type to be dependant on the resulting total map.

A key element of this solution is the type constraint on the Id of the total map produced by the insertion: it guarantees that the new Id is a supertype of the original Id. This is significant and means that all identifiers for the original total map may still be used with the new total map. Scala's support for subtyping and covariance is invaluable here, as it relieves us from explicitly converting all the identifiers we may have stored in other covariant data structures.

2.3 Usage Example

Let us consider a sample application using a simple data model of suppliers, parts and shipments (based on a similar example in [1]). A shipment must refer to both a supplier and a part, and we would like to enforce these two referential integrity constrains.

Listing 6. Suppliers and parts example

```
case class Supplier(name: String, city: String)
case class Part(name: String, weight: Float)
case class Shipment[+S, +P](
    supplierId: S,
    partId: P,
    quantity: Int)

trait SupplierParts { original =>
  val suppliers: Total[Supplier]
  val parts: Total[Part]
  val shipments: List[Shipment[suppliers.Id, parts.Id]]

  def addPart(newPart: Part) =
    new SupplierParts {
      val suppliers: original.suppliers.type = original.suppliers
      val parts = original.parts.insert(newPart).total
      val shipments = original.shipments
    }
}
```

We parameterize the Shipment case class by the identifier types for suppliers and parts, and mark the parameters as covariant. The SupplierParts trait represents an immutable value of the full data model. It enforces the referential integrity constraints by using path-dependent types to refer to the Id types within the suppliers and parts total maps.

The addPart method creates a new SupplierParts with an additional part. No shipment is added here so we want to reuse the original shipment list. Its type refers to the original parts and suppliers total maps. If we are to reuse it, its new type, which refers to the new total maps, must be a supertype. This is guaranteed by the following:

- the new suppliers total map was given the same singleton type as the original, so it has the same Id type,
- the new parts total map has an Id type which is a supertype of the original parts Id type,
- and both List and Shipment are covariant.

2.4 Removal

The removal process works in reverse: the resulting identifier type is a subtype of the original. This implies an additional burden: some old identifiers can no longer serve as identifiers in the new total map. The remove function therefore returns a filter function that may be used to narrow the old identifiers into the new subtype of identifiers.

Listing 7. Removal form a total map

```
sealed trait Total[+V] {
  type Id
  def remove(removedKey: Id): Contraction[Id, V]
}
trait Contraction[K, V] {
  val total: Total[V] {type Id <: K}
  def filter(k: K): Option[total.Id]
}
```

2.5 Related Identifier Types

Refinement types allow us to express that multiple total maps use identical or related identifier types. Assuming we have a total map named t, we could use the following types to represent a total map where the identifiers are a subset, the same set, or a superset.

Listing 8. Related identifier types

```
Total[V] {type Id <: t.Id}
Total[V] {type Id = t.Id}
Total[V] {type Id >: t.Id}
```

A subtype relation between the identifiers of two total maps can be seen as a referential integrity constrain from one identifier (the subtype) to the other. This means that when translating constraints from a relational model, we can also express constrains that originate from a column that is also a primary key.

2.6 Cyclical Constraints

To represent cyclical referential integrity constraints, it will not be possible to express all constraints as path-dependent types without first breaking the cycle. For example we cannot use a single total map to represent an endofunction as **trait** Endo {**val** f: Total[f.Id]}.

Instead we must use a total map to define the domain/-codomain, and a separate one, sharing the same Id type, to define an endofunction that references it.

Listing 9. Endofunction

```
trait Endo {
  val ids: Total[Unit]
  val values: Total[ids.Id] {type Id = ids.Id}
}
```

A similar technique can be used to break longer cycles.

The expression of such cyclical constrains comes at a cost in terms of logical data independence. Adding a field to a data structure could cause a cycle and force a refactoring.

3. Implementation

Our total maps are implemented as binary trees where each internal node can hold one element of the collection. New elements are inserted at the minimum depth where a free node is available. Elements are removed from the node they occupy, and no reorganization of the tree takes place beyond pruning subtrees that are completely empty.

An element with a given identifier is never relocated in the tree. Its identifier is in fact its path from the root node.

3.1 Concrete Identifier Types

The identifier type must characterize the set of paths in the tree that lead to elements of the collection. Our solution is to use a disjunction type with three alternatives and corresponding type parameters. A first type parameter is Unit when the root is an admissible path, and Nothing otherwise. The other two type parameters allow the recursive use of the identifier type to describe valid subpaths.

Listing 10. Identifier types

```
sealed trait AnyId
trait Id2[+U<:Unit, +A1<:AnyId, +A2<:AnyId]
                        extends AnyId
case object Element
                        extends Id2[Unit, Nothing, Nothing]
case class Left[+A1<:AnyId](l: A1)
                        extends Id2[Nothing, A1, Nothing]
case class Right[+A2<:AnyId](r: A2)
                        extends Id2[Nothing, Nothing, A2]
```

An instance of the first alternative Element designates the element at the root of the tree being considered. This constructor is only applicable when U is Unit (when an element is present in the root node). An instance of Left designates an element in the left subtree, and contains the rest of the path as a data member. An instance of Right has a similar role for the right subtree.

Thanks to the covariance annotations of the three type parameters, the identifier types are related by subtyping. Given an identifier type, it is possible to find a supertype that can represent any strict superset of paths. Similarly for all non-empty identifier types, it possible to find a subtype that can represents any strict subset of paths. This enables the implementation of the insertion and removal operation on total maps.

3.2 Integer Identifier Types

As an optimization, we have implemented a different identifier type that is parameterized in the same way as the Id type above, but which uses constant space. It simply stores an integer. This part of the implementation is not type-safe internally, but its soundness can be validated by relating it to the isomorphic Id type shown above.

3.3 Total Map Implementation

Our implementation of the total map sealed trait consists of three subtypes. The EmptyTotal object implements an empty total map, so its identifier type is Nothing. The TotalWith class represents a total map that contains an element in its root, while TotalWithout contains no element at the root. As internal nodes of a binary tree, these two classes also contain two other total maps t1 and t2.

Listing 11. Implementation of identifiers and lookup

```
case object EmptyTotal extends Total[Nothing] {
  type Id = Nothing
  def apply(k: Id): V = k
}
case class TotalWith[+V](v: V, t1: Total[V], t2: Total[V])
    extends Total[V] {
  type Id = Id2[Unit, t1.Id, t2.Id]
  def apply(k: Id): V = k match {
    case Element => v
    case Left(l) => t1(l)
    case Right(r) => t2(r)
  }
}
case class TotalWithout[+V](t1: Total[V], t2: Total[V])
    extends Total[V] {
  type Id = Id2[Nothing, t1.Id, t2.Id]
  def apply(k: Id): V = k match {
    case Left(l) => t1(l)
    case Right(r) => t2(r)
  }
}
```

For EmptyTotal, the apply method will never be called since it takes a Nothing as input. For the internal node classes we simply pattern match on the identifier considering only the admissible cases. When Left or Right is encountered we delegate to the apply function of the appropriate child total map.

Insertion, which is not listed here, proceeds as follows: For EmptyTotal or TotalWithout, we return a TotalWith containing the newly added element. As is required, the Id type of the result will be a supertype. For a TotalWith, we insert in one of its two children. To favor a balanced tree, our approach has been to store the number of total map elements in every node, and select the smallest child.

Most common operations on maps have also been implemented. These implementations follow quite naturally from the strict type signatures.

4. Challenges

We have used the total map collection successfully in various small examples. Overall, the programs remained clean and concise. We did notice some concerns, which may cause problems in the context of a full application. Some of these could possibly be addressed by changes or extensions to the Scala compiler.

4.1 Efficient Narrowing

There are inefficiencies in the filtering process that removes occurrences of identifiers that have been removed from a total map.

First, finding those identifiers rapidly would require indexing. This appears very challenging in a context where type safety must be maintained. We have not explored this.

Second, all data structures that contain identifiers of the old type must be reallocated as part of the filtering operation, even all those that did not in fact contain the removed identifier. Avoiding this seems challenging but possible using checked downcasts.

4.2 Propagation of Identifier Types

The explicit propagation of identifier types through type parameters could become a concern in larger applications. Classes must be parameterized explicitly over the identifier types they contain, even indirectly, and they must bear covariance annotations. However we expect the nesting of data structures to be more limited. In this approach the data models tends to be more relational, so perhaps the propagation of identifier types will not be a significant practical concern.

4.3 Dependent Types in Constructors

As of Scala 2.11.6, it is not possible to declare classes with constructors that have parameters with types that depend on the values of other parameters. This limitation is the subject of an open Scala improvement issue (SI-5712). In this paper we have avoided the problem by directly using type refinements to construct instances of our data model. For larger examples, we can obtain more concise programs by defining functions that create the instances. Once class constructors with dependent parameter types are supported, we will be relieved of this potentially repetitive task.

4.4 Inference of Singleton Types

In the `addPart` method of listing 6, we had to give an explicit singleton type to the `val suppliers`. Inference of the singleton type would have been convenient in this case. This is a minor inconvenience once one understands the problem. However it is likely that many developers would initially be puzzled by this.

5. Future Work

5.1 Persistence

We suspect that the technique presented here can be extended to statically enforce the referential integrity constrains on data persisted within databases. It should be possible to implement a similar total map type which is backed by a database.

To reconcile the mutable nature of most databases and the immutability assumptions of the approach presented here, a database access library could expose a lazily-loaded snapshot of the database at the beginning of a transaction, and consume a similar snapshot at the close of the transaction.

The in-memory data models proposed here are closer to typical relational data models. Similar models and interfaces for in-memory and persisted data would be a step towards to the convenience and reliability potential of orthogonal persistence [3].

5.2 Improving Performance

We also intend to turn our attention to the performance of our implementation. Insertion, removal and lookup are all *O(log(n))* operations, but traversals that also return the identifiers are currently *O(n log(n))* instead of the expected *O(n)*. This is due to the limited set of operations we have exposed on identifiers. We hope to address this limitation in future work, and to publish benchmarks for the total map data structure.

6. Conclusion

We have developed the total map data structure which supports referential integrity constraints that can be directly expressed and enforced by Scala's type system. This has increased our appreciation for Scala's path-dependent types, subtyping and variance annotations. We believe implementing total maps would not be practical or even possible in most other programming languages.

The approach eliminates an important class of errors from programs, and does so in a fairly natural manner. Although we have yet to apply the technique in a real application, the effort involved may be comparable to that involved in weaker techniques based on creating distinct key types. Its main drawback is that removing elements from total maps requires a traversal of data structures that contain identifiers, so that invalid identifiers may be excluded. However, this may be a favourable tradeoff in applications where reliability is a more pressing concern than performance.

The total map data structure is part of a freely available library at: https://github.com/boldradius/total-map. Version 0.2.2 contains the work presented here.

References

[1] C. J. Date. *An Introduction to Database Systems (8th Edition)* 2003: Addison-Wesley.

[2] C. J. Date. 1981. Referential integrity. *In Proceedings of the seventh international conference on Very Large Data Bases - Volume 7* (VLDB '81), Vol. 7. VLDB Endowment 2-12.

[3] Alan Dearle, Graham N. C. Kirby, and Ron Morrison. 2009. *Orthogonal persistence revisited. In Proceedings of the Second international conference on Object databases* (ICOODB'09), Moira C. Norrie and Michael Grossniklaus (Eds.). Springer-Verlag, Berlin, Heidelberg, 1-22.