

Technical Specification

Entity Relations in Bevy

Relational semantics for
the Bevy game engine.

Student 1: Yogesvara Das
Student 2: David Mc Guigan
Project Supervisor: David Sinclair

Student Number: 19712101
Student Number: 19425942

Completion Date: 06/05/2023

0. Table of Contents

0. Table of Contents	2
1. Introduction	3
1.1 Overview	3
1.2 Glossary	3
1.3 Motivation	4
1.4 Research	4
2. High-Level Design	5
2.1 Context Diagram	5
3. Implementation	6
3.1 Relations Implementation	6
3.2 Despawn Policies	7
3.3 Relation Storage	7
3.4 API Implementation	9
3.4.0 Setting Relations	9
3.4.1 Relation Queries	10
4. Problems Solved	14
5. Future Work	15
6. Guidance on Testing	16

1. Introduction

1.1 Overview

The Entity Component System (ECS) architecture is a modern technology gaining popularity in the design of video game engines. It uses a data oriented philosophy which improves both the development experience of programmers as well as the performance of the application itself.

In contrast to Object-Oriented architectures, which rely on the manipulation of classes within large inheritance trees to add data and functionality to a game object, ECS allows data to be separated into small, independent structures which can be freely combined to create entities with complex behaviour.

The aim of our project was to take an existing, open-source game engine, Bevy, and add relational semantics to it, allowing developers to easily express complex relations between entities in a graph-like structure.

As this is a contribution to an open-source project it is important to specify that the only files in the *src* directory which we have written are in the [crates/bevy_ecs/src/relation](https://github.com/bevyengine/bevy/tree/master/crates/bevy_ecs/src/relation) directory.

1.2 Glossary

Entity	A unique identifier typically represented by an unsigned integer that owns zero or more components.
Borrow	Term for indirection in Rust.
Component	Data types that store data for entities.
System	A procedure that queries for components and other world items and performs computation with those items.
Policy	Procedures to execute that uphold a desired property of a relation graph.
Bevy	An open-source ECS game engine written in Rust.
Fragmentation	The splitting up of data corresponding to similar archetypes.
Archetype	Collection of entities that have the same set of components.
Command	Deferred execution mechanism in bevy.
TypeId	Unique identifier for a type in Rust.
Dogfooding	Eating your own dog food or "dogfooding" is the practice of using one's own products or services.

DX	Developer experience.
----	-----------------------

1.3 Motivation

The main motivation behind this project was usability. Relations add the capabilities to express indirection at an ECS level which extends the paradigm to more advanced use cases. They also allow dogfooding the engine to itself which makes offering new first party engine features like user interfaces, render graphs, schedule graphs, soft bodies and much much more very simple. It drastically improves the DX of all current and future contributors.

Both students are also interested in game design and game engine development and took the opportunity to further their experience and knowledge in these domains.

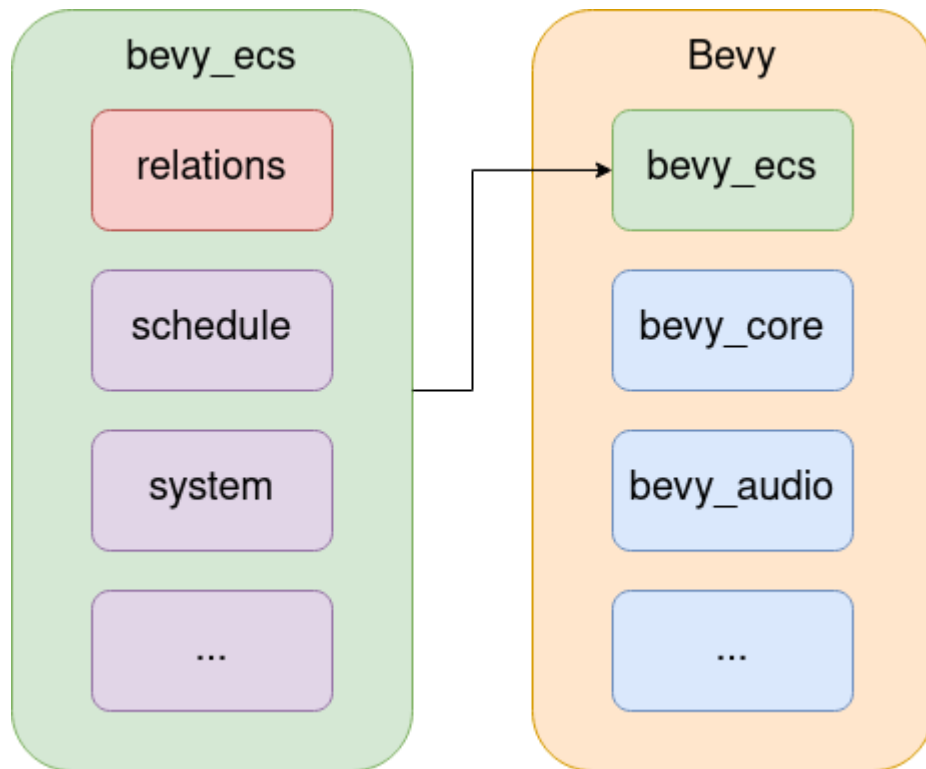
1.4 Research

Many aspects of this project required very involved research from both students.

- Firstly we had to get familiar with Archetypal ECS. There are other kinds of ECS most notably bitset ECS and sparse set ECS but Archetypal ECS is the kind that Bevy uses. Understanding this engine architecture was a prerequisite for understanding how relations fit in and the problems that come with implementing them.
- Secondly we had to get familiar with Bevy. Both students had used bevy before so we were familiar with the API to some extent. We were not familiar with the code base so we took a stab at an onboarding pull request from the backlog before we even started the project.
- Then we had to learn about relations which was mostly through flecs another open source archetypal ECS that features relationships. Bevy has taken inspiration from flecs before. In particular flecs and its creator Sander Mertens is one of the pioneering forces in this space. A great deal of time was spent studying flecs's relations implementation as well as conversing with Sander who was more than happy to answer many of our questions about flecs.
- Finally we looked at some of the existing attempts at relations and what the concerns were. We'd learned archetype fragmentation was a concern as it can adversely affect query construction times and iteration speed. This would be the primary challenge to overcome for our implementation.
- Throughout the project we also had to learn new things as they came up that involved limitations in Rust as well as general graph theory problems.

2. High-Level Design

2.1 Context Diagram



The context diagram shows where our contribution exists in the project. Bevy is composed of almost 40 different crates. One of them is `bevy_ecs`. Our `relations` implementation is a substantial module within `bevy_ecs`. The implementation makes use of the existing features within `bevy_ecs` and can be used as if it were one of those features. No 3rd party additions are required.

3. Implementation

The entire project was implemented in Rust and exists as a submodule within the Bevy engine.

We divided our code into five separate files, each with its own functionality:

- *mod.rs*: Main module file, contains logic for creating and removing relations.
- *joins.rs*: Contains logic for joining relation and entity queries.
- *traversals.rs*: Contains logic for traversing graphs of entity relations.
- *policies.rs*: Contains logic for maintaining graph structure when removing entities and relations.
- *tuple_traits.rs*: Contains the necessary scaffolding for a lot of the type level APIs.

3.1 Relations Implementation

First we needed to provide a way for users to create relations. We did this by providing the *Relation* trait which can be implemented for a struct they define:

```
9 implementations
79  ✓ pub trait Relation: 'static + Sized + Send + Sync {
80      type Storage: ComponentStorage;
81  ✓   const DESPAWN_POLICY: DespawnPolicy = DespawnPolicy::Orphan;
82      const EXCLUSIVE: bool = false;
83  }
84
```

```
11  #[derive(Copy, Clone)]
      3 implementations
12  ✓ pub enum DespawnPolicy {
13      RecursiveDespawn = 0,
14      RecursiveDelink = 1,
15      Reparent = 2,
16      Orphan = 3,
17  }
```

The *EXCLUSIVE* field indicates whether or not an entity can have multiple instances of the relation. The *DESPAWN_POLICY* field indicates the effect that the removal of an entity with this relation will have on the structure of the graph. It can be one of four predefined options. We kept things predefined to reduce complexity.

3.2 Despawn Policies

Our implementation takes advantage of the existing features of Bevy and stores data pertaining to relations in a component on the entity itself.

```
62
63  #[derive(Component, Default)]
64  pub struct Edges {
65      pub(crate) targets: [HashMap<TypeId, HashMap<Entity, usize>>; 4],
66      pub(crate) fosters: HashMap<TypeId, HashSet<Entity>>,
67  }
68
```

Relations are reduced to their `TypeId` when stored so data in the *targets* field is separated based on the despawn policy of each relation which allows for the correct policy to be applied easily.

As seen in the code snippet above, each despawn policy has a precedence with which it is applied. This is necessary to avoid the graph ending up in an invalid state. For example, it would be unacceptable if a reparent policy were applied which reparents an entity to an entity which is then immediately despawned by a despawn policy. By applying the most destructive policies first, we can adjust for their impact on the graph.

There are several ways in which a despawn policy can be triggered and all are accounted for by an enumerator, *Operation*, which gets passed to a function which compiles a list of further operations to be executed.

```
19  pub enum Operation {
20      Despawn(Entity),
21      Delink(Entity, TypeId, Entity), // parent, relation, child
22      Reparent(Entity, TypeId),       // child, relation
23  }
24
```

When executing recursive despawn policies there is the risk of the relation graph being cyclic and causing an infinite loop. Explicitly preventing cyclic graphs would be too expensive an operation as it requires the entire graph to be traversed every time it changes. Instead we opted to keep a `HashSet` of visited nodes when performing recursive operations and ignoring any entities which have already been visited.

3.3 Relation Storage

Alternative implementations of relational ECS, such as flecs, divide archetypes by the relation type as well as the target of said relation. This causes increased fragmentation of archetypes as relations typically have several targets. This fragmentation results in data being divided into more tables and increases the likelihood of cache misses when queries need to access multiple tables in memory, negatively impacting performance.

Below is an example of increased fragmentation caused by multiple targets for the **Likes** relation:

Entity ID	Name	Age	(Likes, e2)	Entity ID	Name	Age	(Likes, e3)
e0	"Frank"	65	0.2	e0	"Frank"	65	0.6
e1	"Charlie"	32	0.1	e1	"Charlie"	32	0.9
		⋮				⋮	

Our implementation avoids this by only fragmenting on the relation type and storing values for all targets within a **Storage<R>** component. The reduction in fragmentation is visible in the example below which contains the same data in a single table.

Entity ID	Name	Age	Edges	Storage<Likes>
e0	"Frank"	65	targets:.... fosters:....	[0.2, 0.6]
e1	"Charlie"	32	targets:.... fosters:....	[0.1, 0.9]
		⋮		

3.4 API Implementation

3.4.0 Setting Relations

Bevy provides an interface for the deferred execution of functions with access to all data within the app known as commands. Relations change archetype information and can also trigger cleanup policies through setting or unsetting so they require deferred execution to prevent unnecessary schedule blocking.

It is through these commands that relations are added and removed.

```
215 pub struct Set<R>
216 where
217     R: Relation,
218 {
219     pub foster: Entity,
220     pub target: Entity,
221     pub relation: R,
222 }
```

```
302 pub struct UnSet<R>
303 where
304     R: Relation,
305 {
306     pub foster: Entity,
307     pub target: Entity,
308     pub _phantom: PhantomData<R>,
309 }
```

3.4.1 Relation Queries

A large part of this project was the type level API which required extensive knowledge of rust's type system. Firstly we had to add a way for queries to specify the relations they wanted to work with. This required adding a new world query type that could accept relations as relations since relations were not components. This is where we encounter the first of our woes. Rust does not have support for variadic generics which means that in order to support all tuple sizes we have to write macros to generate the implementations for any traits that need to take a variable number of type parameters.

```
96 pub trait RelationQuerySet: Sealed + Send + Sync {
97     type Types;
98     type WorldQuery: WorldQuery;
99     type ColsWith<T: Default>: Default;
100 }
101
102 impl<R: Relation> RelationQuerySet for &'_ R {
103     type Types = R;
104     type WorldQuery = StorageWorldQuery<R>;
105     type ColsWith<T: Default> = T;
106 }
107
108 impl<R: Relation> RelationQuerySet for &'_ mut R {
109     type Types = R;
110     type WorldQuery = StorageWorldQueryMut<R>;
111     type ColsWith<T: Default> = T;
112 }
113
114 impl<R: RelationQuerySet> RelationQuerySet for Option<R> {
115     type Types = R::Types;
116     type WorldQuery = Option<R::WorldQuery>;
117     type ColsWith<T: Default> = R::ColsWith<T>;
118 }
119
120 // TODO: All tuple
121 impl<P0: RelationQuerySet> RelationQuerySet for (P0,) {
122     type Types = (P0::Types,);
123     type WorldQuery = (P0::WorldQuery,);
124     type ColsWith<T: Default> = (P0::ColsWith<T>,);
125 }
126
127 impl<P0: RelationQuerySet, P1: RelationQuerySet> RelationQuerySet for (P0, P1) {
128     type Types = (P0::Types, P1::Types);
129     type WorldQuery = (P0::WorldQuery, P1::WorldQuery);
130     type ColsWith<T: Default> = (P0::ColsWith<T>, P1::ColsWith<T>);
131 }
132
133 // TODO:
134 // - Manual `WorldQuery` impl to get `ComponentId` from `World` to remove the usage of `TypeId`
135 // - `TypeId` is not guaranteed to be stable which is a problem for serialization.
136 #[derive(WorldQuery)]
137 #[world_query(mutable)]
138 pub struct Relations<T: RelationQuerySet> {
139     edges: &'static Edges,
140     world_query: T::WorldQuery,
141     _phantom: PhantomData<T>,
142 }
143
```

From here we needed a way to allow the user to specify joins and traversals on the relation types that they queried for.

To allow for the `.join::<R>(q)` and `.breadth_first::<R>(e)` APIs we needed a way to implement the same trait multiple times for the same type. This is not allowed.

BUILD ▶
...
DEBUG ▾
STABLE ▾
...

```

1 pub trait TypedSet<Types, Target> {
2     type Out<Input>;
3     fn set<Input>(self, value: Input) -> Self::Out<Input>;
4 }
5
6 impl<K0, P0> TypedSet<K0, K0> for P0 {
7     type Out<Input> = Input;
8     fn set<Input>(self, value: Input) -> Self::Out<Input> {
9         value
10    }
11 }
12
13 impl<K0, P0> TypedSet<(K0,), K0> for (P0,) {
14     type Out<Input> = (Input,);
15     fn set<Input>(self, value: Input) -> Self::Out<Input> {
16         (value,)
17     }
18 }
19
20 impl<K0, K1, P0, P1> TypedSet<(K0, K1), K0> for (P0, P1) {
21     type Out<Input> = (Input, P1);
22     fn set<Input>(self, value: Input) -> Self::Out<Input> {
23         (value, self.1)
24     }
25 }
26
27 impl<K0, K1, P0, P1> TypedSet<(K0, K1), K1> for (P0, P1) {
28     type Out<Input> = (P0, Input);
29     fn set<Input>(self, value: Input) -> Self::Out<Input> {
30         (self.0, value)
31     }
32 }

```

Execution

Standard Error

Compiling playground v0.0.1 (/playground)

error[E0119]: conflicting implementations of trait `'TypedSet<_, _>, _>'` for type `'(_, _)'`
--> [src/lib.rs:27:1](#)

```

20 | impl<K0, K1, P0, P1> TypedSet<(K0, K1), K0> for (P0, P1) {
    | ----- first implementation here
...
27 | impl<K0, K1, P0, P1> TypedSet<(K0, K1), K1> for (P0, P1) {
    | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ conflicting implementation for '(_, _)'

```

For more information about this error, try `'rustc --explain E0119'`.

error: could not compile `'playground'` due to previous error

This is where the first of our metaprogramming comes in. There is a pattern here in that the differentiating part of each implementation is the position of the **Target** type. So a simple way to work around this is just to add a const generic parameter to the trait. Each const is treated as its own type and with numeric constants we can generate the implementations easily with macros.

BUILD ▶ ... DEBUG ▾ STABLE ▾ ...

```
1 pub trait TypedSet<Types, Target, const POS: usize> {
2     type Out<Input>;
3     fn set<Input>(self, value: Input) -> Self::Out<Input>;
4 }
5
6 impl<K0, P0> TypedSet<K0, K0, 0> for P0 {
7     type Out<Input> = Input;
8     fn set<Input>(self, value: Input) -> Self::Out<Input> {
9         value
10    }
11 }
12
13 impl<K0, P0> TypedSet<(K0,), K0, 0> for (P0,) {
14     type Out<Input> = (Input,);
15     fn set<Input>(self, value: Input) -> Self::Out<Input> {
16         (value,)
17     }
18 }
19
20 impl<K0, K1, P0, P1> TypedSet<(K0, K1), K0, 0> for (P0, P1) {
21     type Out<Input> = (Input, P1);
22     fn set<Input>(self, value: Input) -> Self::Out<Input> {
23         (value, self.1)
24     }
25 }
26
27 impl<K0, K1, P0, P1> TypedSet<(K0, K1), K1, 1> for (P0, P1) {
28     type Out<Input> = (P0, Input);
29     fn set<Input>(self, value: Input) -> Self::Out<Input> {
30         (self.0, value)
31     }
32 }
```

⋮

Execution

Standard Error

Compiling playground v0.0.1 (/playground)
Finished dev [unoptimized + debuginfo] target(s) in 1.25s

Finally we needed a way to allow users to iterate through matched entities in a query for any specified set of operations. Rust has the **Iterator** trait which is responsible for allowing users to loop through collections and write higher order function chains but this was not suitable for our purposes. We need something called “**Lending Iterators**”. Rust’s memory model requires us to uphold special invariants but because we’re creating borrows from borrows we need a special type system feature called Generic Associated Types to express to the type system that we are in fact upholding the necessary invariants.

Unfortunately Generic Associated Types as they exist currently are not capable of expressing Lending Iterators due to type system bugs:

<https://blog.rust-lang.org/2021/08/03/GATs-stabilization-push.html>

After much digging we found a pattern to work around this which is called “lifetime trapping” which makes use of a very advanced and barely usable feature called “higher rank trait bounds” which is the funny looking

for<'r, 'c, ...> syntax seen below:

```
pub trait ForEachPermutations {
    type Components<'c>;
    type Joins<'i, 'a, 'j>;
    fn for_each<Func, Ret>(self, func: Func)
    where
        Ret: Into<ControlFlow>,
        Func: for<'r, 'c, 'i, 'a, 'j> FnMut(
            &'r mut Self::Components<'c>,
            Self::Joins<'i, 'a, 'j>,
        ) -> Ret;
}
```

This pattern makes it so any arguments provided to an anonymous function *cannot leave* the anonymous function. With this we have satisfied the necessary invariants and the type system is happy.

The drawback to this is that users cannot loop over the query with a normal **for** loop and they also lose access to the declarative higher order function constructs **Iterator** provides.

Losing access to **Iterator** also means losing access to the **Iterator** extensions that exist in the Rust ecosystem such as **Itertools** and **Rayon**.

Nevertheless the downsides of this are completely negated. The enabling of this API results in a massive net ergonomic gain.

4. Problems Solved

- We addressed some of the concerns with relations in our storage strategy. Fragmenting only on relation means we trade the satisfiability solver properties fully fragmenting relations bring for better iteration characteristics but this still adds a lot of powerful functionality.
- We broke ground on the type level APIs necessary for relations. This kind of metaprogramming from what we've seen not only doesn't exist in bevy but doesn't exist in any area of the rust ecosystem.
- We addressed ambiguity problems for more complicated relation policies. Unlike flecs which only supports recursive despawning and orphaning we are able to support more types of common cleanup patterns. By creating a system to the policies with precedence based on how much data they alter we not only created a path implementation but also created a way for users to reason about what will happen.
- We originally wanted to do this project on Github because we could get contributor input and participate in the open source aspect more. This was denied for reasons relating to how CA400 is managed. This created a lot of headaches because we have to attribute work that is not ours.

We forked the Bevy repo from github and decided to merge unrelated histories. For the majority of the year we did not have the repo in the structure CA400 demanded because it would make pulling changes from the main bevy repo much more difficult than it should be.

Despite our best efforts whenever we did have to pull changes this created massive merge conflicts which were very tedious and time consuming to fix.

We endured this preventable problem and managed to satisfy the needs of CA400. We restructured the directories in the repo towards the end because at that point what we'd done had stabilised and we didn't need to pull any more changes from bevy main.

5. Future Work

- Use **ComponentId** instead of **TypeId** to type erase. This would be preferred as **TypeId** is not stable and unfriendly to serialisation.
- Write the macros to produce “all tuple” implementations for metaprogramming scaffolding to support all tuple sizes in relation queries. Currently only tuple sizes of arity 2 are supported because that was all we needed to demo and because it was a comfortable size to hand write. We opted not to write macros to produce these during development as they would slow down velocity by adding more complexity to already complex code.
- Remove **CheckedDespawn** and make cleanup implicit. We encountered mysterious unit test failures when changing the existing **Despawn** command directly.

```
failures:
  system::commands::tests::commands

test result: FAILED. 250 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.03s
error: test failed, to rerun pass '-p bevy_ecs --lib'
```

There is also no way to say all exposed and internal APIs for despawning entities were covered with this method either. There are many methods to despawn entities. The only way to be certain of coverage is exhaustively going through the codebase.

- Replace tombstoning strategy for relation storage with something that is better for memory usage. Because of the way the type erasure works we cannot remove relation values in certain cases through cleanup policies. A simple fix would be to change this to a recycling strategy but this would still be unideal. A more sophisticated type erasure strategy is needed which is a non-trivial item of work.

6. Guidance on Testing

Most of the testing was done via unit tests which are functions preceded by the `#[test]` annotation in the project files. We used cargo to run these tests as it is the de facto tool for this in the Rust ecosystem. Additionally we also tested our new feature with our own games essentially doing our own Quality Assurance.

Many projects use what's called the "entity in component" pattern which is just a component that contains an entity value to mimic some of the functionality of relations. It was just a matter of importing our local bevy fork and using our API.

Bevy is a very well organised open source project and has a lot of existing CI on GitHub already. Since this project is a contribution it makes sense to use the CI pipelines they already have in place as that's what will be run when we make this a PR outside of college time. We copied the runners they were using to run our tests on gitlab.

Pipeline



Needs

Jobs

1

Tests

0

Status	Job	Stage
<div><div>✓</div>passed</div>	<div>#91328</div> <div> master  83410cd6</div>	unit-tests