Exploring the Theory and Practice of Concurrency in the Entity-Component-System Pattern

PATRICK REDMOND, University of California, Santa Cruz, USA
JONATHAN CASTELLO, University of California, Santa Cruz, USA
JOSÉ MANUEL CALDERÓN TRILLA, University of Maryland, USA
LINDSEY KUPER, University of California, Santa Cruz, USA

The *Entity-Component-System* (ECS) software design pattern, long used in game development and similar domains, encourages a clean separation of identity (entities), data (components), and computation (systems). Programs written in the ECS pattern are naturally concurrent, and the ECS pattern offers modularity, flexibility, and performance benefits that have led to a proliferation of ECS frameworks. Nevertheless, the ECS pattern is little-known and not well understood outside of a few domains — perhaps because ECS programs can also be difficult to reason about. This problem is compounded because, lacking a formal treatment, the ECS pattern is often explained by way of encoding a concrete problem with the interfaces of a particular ECS framework. We seek a rigorous understanding of the ECS pattern via the design of a formal model, *Core ECS*, that captures the essence of software written in the ECS pattern. We identify a class of Core ECS programs that behave deterministically regardless of scheduling, thus enabling use of the ECS pattern as a deterministic-by-construction concurrent programming abstraction. We then apply our formal model to reflect on the concurrency available in several popular ECS frameworks, using Core ECS to identify gaps in their concurrency offerings, driven by particular performance concerns, that suggest the need for alternative implementation strategies.

1 INTRODUCTION

The *Entity-Component-System* software design pattern (or *ECS pattern*) has been in use for game development since the early 2000s [Bilas 2002; West 2007; Martin 2007a; West 2018] as well as more recently in GUI programming [Raffaillac and Huot 2019] and real-time interactive systems [Hatledal et al. 2021]. Software written in the ECS pattern specifies computations (called *systems*) concerned with an association from identifiers (called *entities*) to domain-specific data values (called *components*). We refer to a program written in this way as an *ECS program*. A primary aim of the ECS pattern is to produce software that is *data-oriented*, letting programmers focus on how objects in a domain relate to each other instead of on control flow [Bilas 2002]. The last decade has seen a proliferation of ECS frameworks for a variety of programming languages [Anderson and Bevy Contributors 2024; Unity Technologies 2024; Mertens 2024a; Schaller 2023; Gillen 2021; Carpay 2018a; Maguire 2018].

The ECS pattern has found a foothold because it affords several benefits. ECS programs are inherently concurrent; the separation of concerns encouraged by the ECS pattern makes it easy to expose parallelism and gain a significant and predictable speedup. The pattern encourages the development of modular, high-level domain-specific primitives, enabling members of a team who are working on different aspects of a project (for example, a game's art assets, networking code, and physics engine) to work independently. Software written using the ECS pattern does not impose a rigid hierarchy of object types, making it easier for developers to adapt to unplanned design changes.

Despite these benefits, the ECS pattern remains common only in the domain of game development, and descriptions of it are rife with the idiosyncrasies of memory layouts and shoehorned interfaces

Authors' addresses: Patrick Redmond, University of California, Santa Cruz, USA; Jonathan Castello, University of California, Santa Cruz, USA; José Manuel Calderón Trilla, University of Maryland, USA; Lindsey Kuper, University of California, Santa Cruz, USA.

that arise from shallowly embedding a programming pattern as a library in some host programming language. To accurately and completely analyze the ECS pattern and its benefits, a model is necessary. Divorced from concerns such as efficient evaluation or mutual exclusion of shared memory, a formal model does not suffer from implementation idiosyncrasies. Being distinct from any particular ECS framework, a formal model of the pattern allows us to better understand and compare the frameworks and even make discoveries that their commonalities obscured. Despite this need, we know of no such formal model. To that end, our goal in this paper is to take the ECS pattern seriously as an object of study, and consider the behavior of ECS programs in a "rigorous and general way" [Hicks 2015]. We developed a core calulus of the ECS pattern by exploring several prominent ECS frameworks, and modeling what we assessed to be their core metaphor. Our work pays off when we find that the ECS pattern is a *deterministic concurrent programming model* with much more concurrency available than contemporary ECS frameworks allow. We make the following series of contributions:

- We present a core calculus of the ECS pattern, *Core ECS*, and a denotational semantics to give it meaning (Section 3). Core ECS captures the essence of the ECS pattern and aids in reasoning about ECS programs. To our knowledge, Core ECS is the first such formalization of the ECS pattern.
- We characterize the concurrency and determinism properties of the ECS pattern using Core ECS as a model (Section 4). We identify that the ECS pattern offers a programming model for *deterministic concurrency*. We define a class of concurrent Core ECS programs which are deterministic in the presence of scheduler non-determinism, suggesting the view that the ECS pattern is a deterministic-by-construction programming abstraction.
- We relate Core ECS to practical implementations of the ECS pattern in frameworks (Section 5). We use Core ECS to make sense of the myriad forms of superficially distinct concurrency that appear in practical frameworks, and find gaps that suggest alternative implementation strategies are desirable.

We provide context for our contributions by introducing the ECS pattern in Section 2. Section 6 discusses related work, and Section 7 concludes.

2 ECS FOR THE UNFAMILIAR

In this section we explain the ECS pattern and terminology for readers who are unfamiliar with it by way of a simple example: a one-dimensional toy physics simulation. Variations of a toy physics simulation are common in articles explaining the ECS pattern and documentation for ECS frameworks. In that same tradition, we have chosen our example to illustrate the essential elements of the pattern as we see them, and will continue to use this example in Section 3 as we develop our formalism. We start this section by giving a very brief intuition for the ECS pattern as a whole, before diving into our example and deeper explanation.

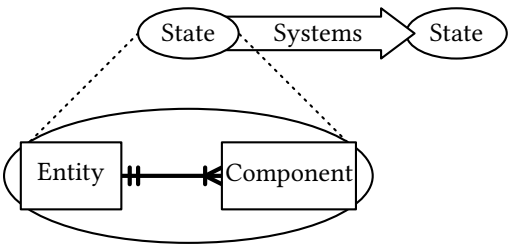


Fig. 1. Entities and their components together comprise the state of a running ECS program. Systems, composed sequentially or concurrently into a single arrow, describe transitions between states.

ECS in brief. Imagine a collection of undifferentiated object identities (entities), each associated with some data properties (components). Entities are characterized only by the selection of components that they have. Now add a fixed set of domain functions (systems) peering into this collection

of entities. Each system takes as its arguments one or more entities based on the precise mix of components the entities have. On some cadence (often in a per-frame "main loop" if this were a videogame), the systems are lined up and called over and over with every combination of entities that match their argument specification. A system's primary task is to modify the components associated with argument entities, create new entities with some associated components, or delete entities along with their components. That's the essence of the ECS pattern. A programmer writes down a fixed set of properties (components) and functions (systems) and then sets up an arbitrary number of entities, each associated with some precisely intentioned set of components. Finally, enter the main loop and quite complex behaviors may quickly arise.

2.1 Entities and components and systems! Oh, my!

The ECS pattern is named for three concepts (visualized in Figure 1) which are repeatedly instantiated in an ECS program.

(1) An *entity* is an identifier value with which component values may be associated.
(2) A *component* is a domain-specific datatype. A value of a component datatype is always associated with an entity (like an object's property). We will usually call such values "components" for brevity, though they are properly "component values".
(3) A *system* is a domain-specific behavior, expressed as a computation over the association of component values and entities. A system typically modifies the associations.

An ECS program is a *fixed set* of components and systems, together with a *varying one-to-many association* of entities to components. From some initial state of the entity-component association, an ECS program proceeds iteratively: The entity-component association determines which systems may be applied to which entities. When applied, a system may mutate the entity-component association. This process repeats by rounds to evolve the program state.

Figure 1 depicts the relationship between entities, components, and systems: An entity-component association is the state of a running ECS program, while the systems, taken together, are its transitions between states. In the rest of this section, we will make these concepts concrete with our physics simulation example.

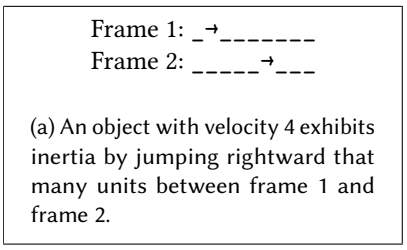
2.2 A toy physics simulation

We consider a toy physics simulation in which objects have a one-dimensional *position* represented by an integer (as if scattered on a number line), and a *velocity*, also represented by an integer (where sign represents the object's direction). The simulation proceeds as a series of frames, one per line. Between each frame, objects in the simulation may move to the left or right, exhibiting *inertia*. For example, in Figure 2a, an object with velocity 4 jumps rightward by 4 units along the line between frame 1 and frame 2 of the simulation. A moving object may also *collide* with a stationary object, annihilating one of them and causing the other to split into two objects traveling in opposite directions at half the original velocity, as in Figure 2b.[1]

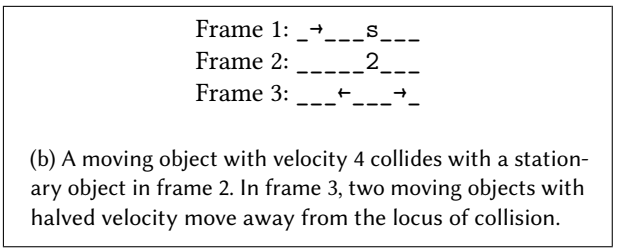
To implement this simulation as an ECS program, we must decide which of its elements correspond to entities, components, and systems. We can make the following representation choices:

- We can represent the simulated objects themselves using entities. Stationary and moving objects will be distinguished by which components they are associated with.
- Since objects have a one-dimensional position and velocity, we can represent those quantities as component values associated with entities. Stationary objects will be those entities having only a position component, and moving objects will be those with both position and velocity.
- We can implement the inertia and collision resolution behaviors using systems.

[1]For this toy simulation, we do not consider collisions between a moving object and another moving object.



(a) An object with velocity 4 exhibits inertia by jumping rightward that many units between frame 1 and frame 2.



(b) A moving object with velocity 4 collides with a stationary object in frame 2. In frame 3, two moving objects with halved velocity move away from the locus of collision.

Fig. 2. Simple text-based visualizations of our toy physics simulation. "←" or "→" indicate an object moving left or right, "s" indicates a stationary object, and "_" indicates empty space. Where two objects occupy the same space, we write the number of objects.

To start the simulation, we initialize it with some number of objects. We create several entities and attach to each a one-dimensional position component. To some subset of the entities, we also attach a nonzero velocity. Frames of the simulation correspond to iterations of the state: To advance to the next frame, determine which systems may be applied to which entities, and apply them to update the entity-component association.

2.3 Queries and Schedules

Our description of the toy physics simulation is necessarily very informal at this stage. Section 3 will give us the language to write down the behavior precisely, and Section 4 will make clear how that behavior is a model of a concurrent program. Nonetheless we wish to prepare the reader by introducing two additional concepts informally: queries and schedules.

System queries. Consider how the inertia and collision resolution behaviors (systems) will specify the entities to which they apply. The inertia system moves objects according to their velocity, and the collision system handles the situation where a moving and a stationary object share the same position. We can phrase these descriptions more precisely in terms of the inclusion or exclusion of components associated with entities (called a *system query*):

- The inertia system applies to any entity with both a position and a velocity component.
- The collision system applies to any pair of entities, where one has both a position and a velocity component, and the other has a position component and *no* velocity component.

When a system query is evaluated against the entity-component association, we call the result its *entity matches*. For the inertia system this could be a set of entities; for the collision system it would be a set of entity tuples, one moving and one stationary according to the query. The entity matches could also be considered to contain the component values associated with each entity that made it a valid part of the result (and we do this in Section 3). For now, we leave the order in which a system processes its entity matches unspecified; Section 4 will return to this question and the role it plays in our model of concurrency.

System schedules. Our description of the physics simulation in Section 2.2 does not describe how the inertia and collision systems will interact with each other, nor does it indicate whether any behaviors are concurrent. We can answer these questions by indicating how each individual system is treated and how distinct systems relate to each other (and this is called a *system schedule*):

- The inertia system runs concurrently, and afterward, the collision system runs sequentially.

This is only one possible schedule for our systems. Instead of running one after the other, a schedule could specify that two systems run concurrently. However in this example we hope the

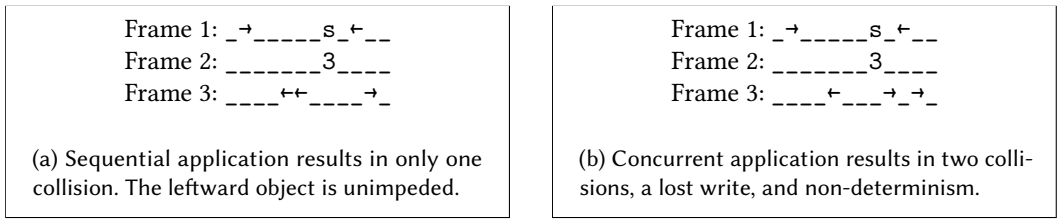


Fig. 3. A rightward-moving object travels at velocity 6 and a leftward-moving object travels at velocity -2. Both are set to collide with a stationary object, but there is more than one way to run the collision system, resulting in different states by frame 3.

reader will recognize that, (1) changing the schedule to run the two systems concurrently, or (2) changing only the collision system to run concurrently, are both options which lead to undesirable non-determinism! Consider the case where two moving objects share a position with a single stationary object, perhaps because they approached from either side, as in Figure 3. Our chosen schedule results in the scenario of Figure 3a, but taking the second option may result in the scenario of Figure 3b.[2] We discuss these scenarios while introducing our formalism in Section 3, and make clear how our formalization models concurrency trade-offs in Section 4.

3 CORE ECS

In this section we describe our core calculus for the ECS pattern, *Core ECS*, and provide a formal denotational semantics of Core ECS to give meaning to the ECS pattern.[3] We discuss how the features of Core ECS relate to practical ECS frameworks in Section 5.

E	Entity identifier type
K	Component schema
I	Component label set

Fig. 4. Symbolic parameters to Core ECS. K is indexed by $i \in I$. Each K_i is a type in the sub-language, as is E.

In Core ECS all *state* is represented by an association between entity identifiers and component values (Section 3.1). A Core ECS program is a *schedule* of systems (Section 3.6) describing how to decompose and update that state. Each *system* is a pair of a query vector and a function that produces a mask (Section 3.5). A single *query* against state specifies a set of entities based on whether they have certain components (Section 3.2). A *query vector* acts as a system query, defining the input to a system as the cross of the results of each query in the vector, and yielding a collection of *entity matches* (Section 3.3). A *mask* describes how to add, update, or remove, entity-component associations in the state (Section 3.4).

3.1 Parameters, Grammars, and State

Parameters. To describe an ECS program we will need a programming language with which to write down systems. We avoid fully integrating Core ECS and its semantics with a specific language; instead Core ECS is parameterized by an underlying programming language (or sub-language). We will need types from the underlying programming language to use as the entity identifiers and components that will populate the state of the ECS program. Therefore Core ECS

[2]We assume that the collision system always destroys the moving object. If such a system were run concurrently with itself, it will allow both moving objects to collide with the stationary object. The choice of which object collides first is non-deterministic, and that collision's write to the velocity of the stationary object will be lost.

[3]We have implemented Core ECS in Haskell, but nothing about the design of Core ECS is Haskell-specific and no knowledge of Haskell is required to read this section.

$$
\begin{array}{rcll}
q : Q & ::= & \mathrm{incl}_i \mid \mathrm{excl}_i \mid \mathrm{anyway}_i \mid (q \wedge q') & \text{Query} \\
m : M & ::= & \mathrm{attach}_i(e, k_i) \mid \mathrm{detach}_i(e) \mid (m \bullet m') \mid \nu(\ E \to M\ ) \mid \mathrm{nil} & \text{Mask} \\
s : S & ::= & (\ \vec{q},\ E^{\dim(\vec{q})} \times (\!|\vec{q}|\!) \to M\ ) & \text{System} \\
z : Z & ::= & \mathrm{conc}(s) \mid \mathrm{seq}(s) \mid (z \parallel z') \mid (z \fatsemi z') & \text{Schedule}
\end{array}
$$

Fig. 5. Grammars defining how to construct Core ECS terms. Types written with a grey background indicate terms of that type in the underlying language.

is also parameterized by a type E and a collection of component types K. The component types come to us in a mapping from component label $i \in I$ to component type K_i, which we will call the program *schema*. We will use e and k_i to refer to terms of the types E and K_i, respectively. Figure 4 summarizes the parameters of Core ECS to which we assign a symbol (the sub-language has none).

Figure 7 shows examples of parameters E and K for the toy physics simulation of Section 2.2, leaving parameter I implied by the domain of the chosen K. Throughout the rest of the paper, we will use such boxed figures continue the running example of the toy physics simulation.

Since Core ECS characterizes a programming *pattern*, most any underlying programming language is an appropriate choice as long as it supports a few key features (functions, sums, products, and a unit, whose types we will write with $\to$, $+$, $\times$, and $\top$, respectively). The sub-language must also have a way to generate fresh (universally unique) entity identifiers and a way to communicate masks (M in Figure 5) to Core ECS. Although we give examples of Core ECS programs that fix a sub-language, our specification of Core ECS itself cannot refer to terms of the underlying language directly because it is a parameter; instead we will write a type with a grey background (e.g. $E \to M$) to stand in for a term in the underlying language having that type.

Grammars. Figure 5 summarizes the grammars used to construct a Core ECS term; later subsections will go into more detail. For now, know that: A Core ECS program is a *schedule* of *systems*. Each system is a pair of a *query vector* and a function in the underlying programming language that produces a *mask*. A system function takes as input a pair, for which type of the second component is dependent on the query vector according to the function $(\!|-|\!)$ in Figure 6.

$$
\begin{aligned}
(\!|\mathrm{incl}_i|\!) &\triangleq K_i \\
(\!|\mathrm{excl}_i|\!) &\triangleq \top \\
(\!|\mathrm{anyway}_i|\!) &\triangleq K_i + \top \\
(\!|q \wedge q'|\!) &\triangleq (\!|q|\!) \times (\!|q'|\!)
\end{aligned}
\qquad
(\!|\vec{q}|\!) \triangleq \prod_{1 \le j \le \dim(\vec{q})} (\!|\vec{q}_j|\!)
$$

Fig. 6. Query and query-vector result type.

We write $\vec{\ }$ over terms indicate a vector and we use $\dim(-)$ for a vector's size. We write T^n for the type of a vector of size n containing elements of type T. Figure 5 uses all three of these conventions; in particular, the type of the function in a system takes $E^{\dim(\vec{q})}$ as part of its input, which is a vector of size $\dim(\vec{q})$ that contains entities.

State. All information about the progress of a running ECS program in Core ECS is represented by a single logical "entity-component association". In this *state* an entity may be associated with at most one value of each component type. We structure Core ECS state as an indexed family of partial mappings from entities to component values. Given parameters E, K, and I, state c is indexed by $i \in I$, and each c_i is a partial mapping with type $E \rightharpoonup K_i$. Figure 7 demonstrates a state for our running example, and we give the definition of state below at left.

$$
c \triangleq \{c_i : E \rightharpoonup K_i\}_{i \in I}
\qquad\qquad
\mathrm{live}(c) \triangleq \bigcup_{i \in I} \mathrm{dom}(c_i)
$$

$$
\begin{aligned}
E &\triangleq \{e_1, e_2, e_3, \dots\} \\
K &\triangleq \{\mathbf{Pos} \mapsto \mathbb{Z}, \mathbf{Vel} \mapsto \mathbb{Z}\} \\
c &\triangleq \{\mathbf{Pos} \mapsto \{e_1 \mapsto 1, e_2 \mapsto 7, e_3 \mapsto 9\}, \mathbf{Vel} \mapsto \{e_1 \mapsto 6, e_3 \mapsto -2\}\}
\end{aligned}
$$

Fig. 7. We represent the state c of the physics simulation from Frame 1 of Figure 3a by defining an entity type E and schema K: We choose a set of opaque entity identifiers for E, and a K that maps the opaque component labels **Pos** and **Vel** to the type of integers. In the example state c we have two moving objects represented by entities e_1, e_3, and one stationary object e_2.

At any point during the execution of an ECS program, it is helpful to know the set of entities that exist in at least one partial mapping of the state, because they have at least one component attached. This set of live entities $\mathrm{live}(c)$ for state c, defined above at right, is the union of the domains of the partial mappings (writing $\mathrm{dom}(-)$ for the domain of a mapping). Only live entities are considered by system queries for inclusion in entity matches. Attaching a single component to an entity is sufficient for it to appear in the set of live entities, and conversely, detaching all components is necessary to remove the entity.

3.2 Queries

A system specifies its input using a simple query language to identify a subset of live entities. A system may have more than one query, meaning that it takes more than one entity at a time as input (Section 3.3), but here we focus on single queries. A single query describes a single entity in terms of its components, but may have many matches among the live entities. A query indicates not only the entity, but also which of its components, to provide to a system. The grammar of queries given in Figure 5 is repeated here.

$$q : Q \quad ::= \quad \mathrm{incl}_i \mid \mathrm{excl}_i \mid \mathrm{anyway}_i \mid (q \wedge q')$$

A query is a conjunction $(q \wedge q')$ of constraints (incl_i and excl_i) that filter entities based on whether each has an association with the component indicated by the label i (except for anyway_i, which performs no filtering).

The meaning of a query, with respect to a state c, is given by the function $[\![-]\!]_c$ defined below at left. This function retrieves a partial mapping of entities matching the query. The value associated with each entity in the mapping, its *component result*, is made up of those component values selected by the query. The type of the component result is given for a query by the function $(\!|-|\!)$, shown in Figure 6 and duplicated below at right. For a query q, the result $[\![q]\!]_c$ will have type $E \mathrel{\dot{\to}} (\!|q|\!)$. See Figure 8 for an interpretation of the queries introduced in Section 2.3.

$$
\begin{aligned}
[\![\mathrm{incl}_i]\!]_c &\triangleq c_i & (\!|\mathrm{incl}_i|\!) &\triangleq K_i \\
[\![\mathrm{excl}_i]\!]_c &\triangleq \{*\}_{e \in \mathrm{live}(c) \setminus \mathrm{dom}(c_i)} & (\!|\mathrm{excl}_i|\!) &\triangleq \top \\
[\![\mathrm{anyway}_i]\!]_c &\triangleq c_i \mathbin{\dot\cup} \{*\}_{e \in \mathrm{live}(c) \setminus \mathrm{dom}(c_i)} & (\!|\mathrm{anyway}_i|\!) &\triangleq K_i + \top \\
[\![q \wedge q']\!]_c &\triangleq \{([\![q]\!]_c(e), [\![q']\!]_c(e))\}_{e \in \mathrm{live}(c)} & (\!|q \wedge q'|\!) &\triangleq (\!|q|\!) \times (\!|q'|\!)
\end{aligned}
$$

The incl_i constraint retrieves c_i, the partial mapping of entities that have the indicated component, mapped to the component values themselves. The excl_i constraint retrieves the complement of incl_i, every live entity that does not have the indicated component. Since there is no associated component value for this mapping, every entity is mapped to $*$ (unit) with type $\top$. We write $a \setminus b$ for the members of a not present in b. The anyway_i constructor retrieves a mapping containing *every*

$$
\begin{aligned}
x &\triangleq \mathrm{incl}_{\mathbf{Pos}} \wedge \mathrm{incl}_{\mathbf{Vel}} \\
(\!|x|\!) &= (\!|\mathrm{incl}_{\mathbf{Pos}}|\!) \times (\!|\mathrm{incl}_{\mathbf{Vel}}|\!) \\
&= K_{\mathbf{Pos}} \times K_{\mathbf{Vel}} \\
&= \mathbb{Z} \times \mathbb{Z} \\
[\![x]\!]_c &= \{(c_{\mathbf{Pos}}(e), c_{\mathbf{Vel}}(e))\}_{e \in \{e_1, e_2, e_3\}} \\
&= \{e_1 \mapsto (1, 6), e_3 \mapsto (9, -2)\}
\end{aligned}
$$

(a) Query x retrieves entities with both position and velocity: the moving objects.

$$
\begin{aligned}
y &\triangleq \mathrm{incl}_{\mathbf{Pos}} \wedge \mathrm{excl}_{\mathbf{Vel}} \\
(\!|y|\!) &= (\!|\mathrm{incl}_{\mathbf{Pos}}|\!) \times (\!|\mathrm{excl}_{\mathbf{Vel}}|\!) \\
&= K_{\mathbf{Pos}} \times \top \\
&= \mathbb{Z} \times \top \\
[\![y]\!]_c &= \{(c_{\mathbf{Pos}}(e), \{e_2 \mapsto *\}(e))\}_{e \in \{e_1, e_2, e_3\}} \\
&= \{e_2 \mapsto (7, *)\}
\end{aligned}
$$

(b) Query y retrieves entities with position but without velocity: the stationary objects.

Fig. 8. Two example queries with applications of $[\![-]\!]_c$ and $(\!|-|\!)$ against the state given in Figure 7.

$$
\begin{aligned}
(\!|\langle x, y \rangle|\!) &= (\!|x|\!) \times (\!|y|\!) \\
&= (K_{\mathbf{Pos}} \times K_{\mathbf{Vel}}) \times (K_{\mathbf{Pos}} \times \top) \\
&= (\mathbb{Z} \times \mathbb{Z}) \times (\mathbb{Z} \times \top) \\
[\![\langle x, y \rangle]\!]_c &= [\![x]\!]_c \times [\![y]\!]_c \\
&= \{e_1 \mapsto (1, 6), e_3 \mapsto (9, -2)\} \times \{e_2 \mapsto (7, *)\} \\
&= \{\langle e_1, e_2 \rangle \mapsto \langle (1, 6), (7, *) \rangle, \\
&\quad\ \langle e_3, e_2 \rangle \mapsto \langle (9, -2), (7, *) \rangle\}
\end{aligned}
$$

Fig. 9. Entity matches for query vector $\langle x, y \rangle$ consisting of queries from Figure 8, interpreted over state from Figure 7. Together, these queries are the query vector of the collision resolution system described in Section 2.

live entity, that yields the component if it is present and the unit value otherwise.[4] The conjunction of constraints behaves like a relational join: the result sets of the conjuncts are filtered to those entities matched by both queries, and the component results retrieved for each query are combined in a tuple.

3.3 System query: Query vector to Entity matches

A system query is specified with a *query vector* and its resultant entity matches is a combination of those queries' results. Systems that implement interactions between entities (e.g. the collision system of Section 2.3) use a query vector with multiple queries, and take as input an entity match containing that many entities and the same number of component results. The entity matches for a system query vector are the cartesian product of the results of its constituent queries. To compute the entity matches for a system, we extend $[\![-]\!]_c$ to take a query vector $\vec{q}$ and return a product of the partial mappings, below at left.

$$
[\![\vec{q}]\!]_c \triangleq \prod_{1 \le j \le \dim(\vec{q})} [\![q_j]\!]_c \qquad\qquad (\!|\vec{q}|\!) \triangleq \prod_{1 \le j \le \dim(\vec{q})} (\!|q_j|\!)
$$

Recall from Section 3.2 that the type of a query result is a partial mapping. This is still true for a vector of queries: The cartesian product of n query results is a single partial mapping of type

[4] The anyway_i constructor does not play an important role in the rest of this paper. However, in practice it is common for an ECS program to deal with optional components, and so we include it for completeness.

$E^n \dashrightarrow (\!|\vec{q}|\!)$ that takes a vector of n entities $\vec{e}$ to a vector of n component results $\vec{w}$. The jth entity and jth component result in each entity match $\vec{e} \mapsto \vec{w}$ are drawn from the result of the jth query in the query vector $\vec{q}$. Figure 9 demonstrates the computation of entity matches for the example collision system. We will also find it useful in Section 3.5 to regard any partial mapping produced by $[\![-]\!]_c$ as having some consistent but unspecified total order as in $\{\vec{e} \mapsto \vec{w},\ \vec{e'} \mapsto \vec{w'},\ \ldots\}$.

3.4 Masks

Given an entity match as input, a system specifies how Core ECS state should change by generating a *mask*. Each system includes a function written in the programming language over which Core ECS is parameterized, and it is the task of this function to produce the mask. Intuitively, a mask is a description of a state update; it replaces some of the mappings with updated values, others are made undefined, and some new mappings may be added. The grammar for masks first shown in Figure 5 is as follows.

$$m : M \quad ::= \quad \text{attach}_i(e, k_i) \ \mid\ \text{detach}_i(e) \ \mid\ (m \bullet m') \ \mid\ \nu(\ E \to M\ ) \ \mid\ \text{nil}$$

The second parameter of the $\text{attach}_i(e, -)$ constructor is a value k_i of the indicated component i. The $\nu(-)$ "new" form requires a function written in the underlying language (see Figure 10 for an example). The composition operator $m \bullet m'$ is right-biased, and nil is an empty mask.

The meaning of a mask m interpreted against state c is a new state given by the function $c \downarrow m$ defined at left. We use the notation $t\{k \mapsto v\}$ for the mapping t in which k has been updated to the value v, and $t\{k \mapsto \bot\}$ for the mapping t in which k has been made undefined.

$$\begin{aligned} c \downarrow \text{attach}_i(e, k_i) &\triangleq c\{i \mapsto c_i\{e \mapsto k_i\}\} \\ c \downarrow \text{detach}_i(e) &\triangleq c\{i \mapsto c_i\{e \mapsto \bot\}\} \\ c \downarrow (m \bullet m') &\triangleq (c \downarrow m) \downarrow m' \\ c \downarrow \nu(f) &\triangleq c \downarrow f(\textbf{fresh}) \\ c \downarrow \text{nil} &\triangleq c \end{aligned}$$

The $\text{attach}_i(e, k_i)$ and $\text{detach}_i(e)$ masks update state c such that i maps to c_i, itself updated with a new mapping for entity e. Attach will add or overwrite the mapping for entity e to the component value k_i. Detach will remove the mapping for e from c_i so that it is undefined. As discussed in Section 3.1, we assume a facility for generating never-before-seen entities, referenced here as **fresh**, and used to handle the $\nu(-)$ case. Both the $\nu(f)$ and $m \bullet m'$ forms use the $\downarrow$ function recursively. For $\nu(f)$ the environment provides a fresh entity, which is passed into the function f to produce a mask. The composition $m \bullet m'$ operator first applies m and next m', such that coincident mappings in m will be replaced by those in m'.

We demonstrate the interpretation of a mask in Figure 10 using the vocabulary of our running example from Section 2.2. This demonstration also introduces the underlying language that we will use throughout our subsequent examples. This language is a parameter to Core ECS, employed only for demonstration: It is a lambda calculus extended with conveniences to make writing the example programs concise (numbers, booleans, if-then-else, tuple and vector patterns; see the appendix for the full grammar). As it is not the focus of our paper, we do not provide a semantics for it — its behavior is standard and contains no surprises.

3.5 Systems

With our discussion of system inputs in Sections 3.2 and 3.3, and system outputs in Section 3.4, we can now explain Core ECS's notion of a system and how it uses entity matches. We reproduce the definition from Figure 5 here.

$$s : S \quad ::= \quad (\ \vec{q},\ E^{\dim(\vec{q})} \times (\!|\vec{q}|\!) \to M\ )$$

$$
\begin{aligned}
& c \downarrow (\text{detach}_{\mathbf{Pos}}(e_1) \bullet \nu(\lambda\, d.\ \text{attach}_{\mathbf{Vel}}(d, 2))) \\
&= (c \downarrow \text{detach}_{\mathbf{Pos}}(e_1)) \downarrow \nu(\lambda\, d.\ \text{attach}_{\mathbf{Vel}}(d, 2)) \\
&= \{\mathbf{Pos} \mapsto \{e_2 \mapsto 7, e_3 \mapsto 9\}, \mathbf{Vel} \mapsto \{e_1 \mapsto 6, e_3 \mapsto -2\}\} \downarrow \nu(\lambda\, d.\ \text{attach}_{\mathbf{Vel}}(d, 2)) \\
&= \{\mathbf{Pos} \mapsto \{e_2 \mapsto 7, e_3 \mapsto 9\}, \mathbf{Vel} \mapsto \{e_1 \mapsto 6, e_3 \mapsto -2\}\} \downarrow \text{attach}_{\mathbf{Vel}}(e_4, 2) \\
&= \{\mathbf{Pos} \mapsto \{e_2 \mapsto 7, e_3 \mapsto 9\}, \mathbf{Vel} \mapsto \{e_1 \mapsto 6, e_3 \mapsto -2, e_4 \mapsto 2\}\}
\end{aligned}
$$

Fig. 10. An example of a mask interpreted against the state in Figure 7. It is only illustrative — the resulting state isn't used in our running example (and is nonsensical, anyway, as e_1 ends with velocity but no position).

$$
\begin{aligned}
\beta &\triangleq q_\beta,\ f_\beta \\
q_\beta &\triangleq \langle x \rangle \\
f_\beta &\triangleq \lambda\, (e_j,\ (p_j, v_j))\,. \\
&\quad \text{attach}_{\mathbf{Pos}}(e_j, p_j + v_j)
\end{aligned}
$$

(a) The inertia system β updates the position of moving entity e_j by adding in one unit of its velocity v_j (reflecting a fixed time gap).

$$
\begin{aligned}
\delta &\triangleq q_\delta,\ f_\delta \\
q_\delta &\triangleq \langle x, y \rangle \\
f_\delta &\triangleq \lambda\, (\langle e_j, e_h \rangle,\ \langle (p_j, v_j), (p_h, -) \rangle)\,. \\
&\quad \textbf{if } p_j \stackrel{?}{=} p_h \textbf{ then} \\
&\qquad \text{detach}_{\mathbf{Pos}}(e_j) \bullet \text{detach}_{\mathbf{Vel}}(e_j) \bullet \\
&\qquad \text{attach}_{\mathbf{Vel}}(e_h, \lceil v_j/2 \rceil) \bullet \\
&\qquad \nu\, (\lambda\, e_\ell.\ \text{attach}_{\mathbf{Pos}}(e_\ell, p_j) \bullet \text{attach}_{\mathbf{Vel}}(e_\ell, \lceil -v_j/2 \rceil)) \\
&\quad \textbf{else } \text{nil}
\end{aligned}
$$

(b) The collision resolution system δ splits the velocity of a moving entity e_j, when it collides with a stationary entity e_h, among the unmoving entity and a new entity, e_ℓ.

Fig. 11. We formalize the systems from Section 2.2 with the queries from Figures 8 and 9 by writing down terms in the language defined for examples.

A system is a static pairing of a query vector and a function in the underlying programming language. For a system s, system query q_s refers to the query vector and system function f_s is the function written in the underlying language. The function f_s takes an entity match of the type produced by q_s, and returns a mask describing how to update state.

Figure 11 shows Core ECS definitions for the inertia and collision resolution systems first described in Section 2.2. Recall from Section 2.3, that the collision system (δ in Figure 11b) applies to any pair a moving and a stationary object. We made this query vector, and its result, concrete in Section 3.3 by describing that an entity match is an element of the cartesian product of query results from a query vector, and working out that result in Figure 9. Now we can see here in Figure 11b that, not only does system query q_δ have those two elements, the inputs to system function f_δ (the entity vector and component result vector) also have two elements, all corresponding to the moving and stationary object contained in each entity match.

We will now define precisely how those entity matches are divided and applied to a system function. We will introduce two different ways to produce a mask for a state using a system: *concurrent production* and *sequential production*. Both use a common notion of applying a system to entity matches, but differ in which entity matches are applied. We must first define two necessary utilities, *applying* a system to entity matches to produce a composite mask, and *rolling* a system over

entity matches to produce a new entity matches. With those definitions: concurrent prouduction simply applies a system to entity matches to produce a mask, whereas sequential production first rolls the system over the entity matches and then applies the system.

The definition of how to *apply* a system s to entity matches is given below. Recall from Section 3.3 that entity matches have a consistent but unspecified total order as in $\{\vec{e}_1 \mapsto \vec{w}_1,\ \vec{e}_2 \mapsto \vec{w}_2,\ ...\}$. We use that ordering here to write a schematic definition of function application syntax against the name of the system, $s(-)$.

$$s(\{\vec{e}_1 \mapsto \vec{w}_1,\ \vec{e}_2 \mapsto \vec{w}_2,\ ...\}) \triangleq f_s(\vec{e}_1, \vec{w}_1) \bullet f_s(\vec{e}_2, \vec{w}_2) \bullet ...$$

That is, applying a system to some entity matches *en masse* is the same as composing the masks generated by applying the system function to each match one at a time. Since the order of the entity matches is unspecified, so too is the order of composition of the resulting masks. In Section 4 we will say more about the implications of this unspecified order.

We define how to *roll* a system s over entity matches r from a starting state c, next. The function $\text{roll}_s(c, r)$ replaces or drops elements of r to account for the cumulative effect of masks over c generated by s while passing through r.[5] It is called "roll" by analogy with the rolling shutter of a camera, in which each line of pixels is observed a slight amount of time after the line prior.

$$\text{roll}_s(c, \{\vec{e} \mapsto \vec{w},\ ...r\}) \triangleq \textbf{let } \vec{w}' = [\![q_s]\!]_c(\vec{e}) \textbf{ in } \{\vec{e} \mapsto \vec{w}',\ ...\text{roll}_s(c \downarrow f_s(\vec{e}, \vec{w}'), r)\}$$
$$\text{roll}_s(c, \emptyset) \triangleq \emptyset$$

Roll inductively visits each of the given entity matches following their order $\{\vec{e}_1 \mapsto \vec{w}_1, \vec{e}_2 \mapsto \vec{w}_2,\ ...\}$. Roll performs the system query $[\![q_s]\!]_c$, at given state c, and looks up $\vec{e}$ in the resulting entity matches to obtain an updated component result vector $\vec{w}'$. In the new entity matches that roll returns, $\vec{e}$ is mapped to the updated $\vec{w}'$, and the recursive step uses state updated with the mask $f_s(\vec{e}, \vec{w}')$. As a notational convenience in this definition, when $[\![q_s]\!]_c$ is undefined on $\vec{e}$ (meaning $\vec{w}'$ is $\perp$) the system function f_s is considered to yield nil, and we regard $\vec{e}$ as dropped from the resulting entity matches (the rest of the entity matches r are processed normally). This elision is what underlies the behavior of Figure 3a in which only one collision occurs even though two appear to be eligible.

We now define concurrent and sequential production of a mask for a state using a system. An illustration of sequential production appears in Figure 12.

Concurrent production	Sequential production
$s([\![q_s]\!]_c)$	$s(\text{roll}_s(c, [\![q_s]\!]_c))$

Concurrent production observes the state c once to obtain and apply entity matches to the system, such that each mask over c produced by f_s is not visible to the other uses of f_s. In contrast, sequential production observes state once to obtain initial entity matches and again before processing each, such that each mask produced by f_s is in light of every prior mask produced by f_s.

3.6 Schedules

We complete our description of Core ECS by introducing how one or more systems combine in a *schedule* to make a complete Core ECS program. A schedule precisely describes when each system query vector observes state and when each mask produced by a system is applied, all relative to each other. The grammar of schedules is as follows.

$$z : Z \quad ::= \quad \text{conc}(s) \ \mid\ \text{seq}(s) \ \mid\ (z \parallel z') \ \mid\ (z \fatsemi z')$$

The $\text{conc}(s)$ and $\text{seq}(s)$ constructors lift a single system to a schedule. The $(z_1 \parallel z_2)$ and $(z_1 \fatsemi z_2)$ constructors compose sub-schedules. Both pairs include a concurrent and a sequential variant, and

[5] Many readers will recognize this as a fold-left with an inlined function to filter and map the input structure.

$$
\begin{aligned}
c' &\triangleq \{\mathbf{Pos} \mapsto \{e_1 \mapsto 7, e_2 \mapsto 7, e_3 \mapsto 7\}, \mathbf{Vel} \mapsto \{e_1 \mapsto 6, e_3 \mapsto -2\}\} \\
\delta(\mathrm{roll}_\delta(c', [\![q_\delta]\!]_{c'})) &= \delta(\mathrm{roll}_\delta(c', \{\langle e_1, e_2\rangle \mapsto \langle (7,6), (7,*)\rangle, \langle e_3, e_2\rangle \mapsto \langle (7,-2), (7,*)\rangle\})) \\
&= \delta(\{\langle e_1, e_2\rangle \mapsto \langle (7,6), (7,*)\rangle\}) \\
&= f_\delta(\langle e_1, e_2\rangle, \langle (7,6), (7,*)\rangle) \\
&= \mathrm{detach}_{\mathbf{Pos}}(e_1) \bullet \mathrm{detach}_{\mathbf{Vel}}(e_1) \bullet \mathrm{attach}_{\mathbf{Vel}}(e_2, \lceil 6/2 \rceil) \bullet \\
&\quad \mathrm{attach}_{\mathbf{Pos}}(e_4, 7) \bullet \mathrm{attach}_{\mathbf{Vel}}(e_4, \lceil -6/2 \rceil)
\end{aligned}
$$

Fig. 12. Sequential production of a mask using collision system δ (Figure 11b) for state c'. In state c', two moving objects share position with one stationary object. State c' was obtained (not shown) by applying to state c (Figure 7) the concurrent production mask for state c using inertia system β (Figure 11a).

we will explore the implications of the concurrent variants for the determinism of a Core ECS program in Section 4.

We define the function $c \Downarrow z$, below, to interpret a schedule z over a state c and produce a composite mask which may be used to advance the state of an ECS program.

$$
\begin{aligned}
c \Downarrow \mathrm{conc}(s) &\triangleq s([\![q_s]\!]_c) \\
c \Downarrow \mathrm{seq}(s) &\triangleq s(\mathrm{roll}_s(c, [\![q_s]\!]_c)) \\
c \Downarrow (z \parallel z') &\triangleq (c \Downarrow z) \bullet (c \Downarrow z') \\
c \Downarrow (z \fatsemi z') &\triangleq (c \Downarrow z) \bullet ((c \downarrow (c \Downarrow z)) \Downarrow z')
\end{aligned}
$$

The cases for $c \Downarrow \mathrm{conc}(s)$ and $c \Downarrow \mathrm{seq}(s)$ correspond to the concurrent production and sequential production of a mask using system s, as discussed in Section 3.5. Recall that concurrent production observes state only once, whereas sequential production observes it before every call to f_s. The cases for $c \Downarrow (z \parallel z')$ and $c \Downarrow (z \fatsemi z')$ have a similar relationship to each other. In the $c \Downarrow (z \parallel z')$ case the sub-schedules are treated as concurrent; the mask generated by z is not visible in z' and vice versa. Whereas the $c \Downarrow (z \fatsemi z')$ case executes sub-schedules in sequence, such that the mask generated by z is applied to c and visible to z' (but not vice versa).

To advance the state of a Core ECS program: Interpret the schedule z over the current state c, as in $c \Downarrow z$, to obtain a mask representing the current state transition. Next interpret that mask m over state c, as in $c \downarrow m$, to obtain updated state c'. We define schedule application $z(c)$ below, as a shorthand for this process, by borrowing function application syntax.

$$z(c) \triangleq c \downarrow (c \Downarrow z)$$

A Core ECS "main loop" could be reasonably defined with schedule application as its central activity.[6] We illustrate an interpretation of a schedule in Figure 13.

4 CONCURRENCY IN ECS

In this section we characterize the concurrency and determinism properties of the ECS pattern using Core ECS as a model. We identify the elements of Core ECS that are notionally concurrent, and that under scheduler non-determinism could lead to non-deterministic results. We identify a new ECS-based model for deterministic concurrency by using Core ECS to define a class of ECS programs that are deterministic despite scheduler non-determinism.

[6]We have a faithful implementation of Core ECS for which the main loop applies the schedule in each iteration.

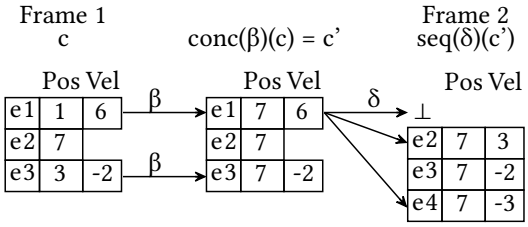
$$
\begin{aligned}
z &\triangleq \mathrm{conc}(\beta) \fatsemi \mathrm{seq}(\delta) \\
z(c) &= c \downarrow (c \Downarrow z) && \text{Substitute definition of } z. \\
&= c \downarrow (c \Downarrow (\mathrm{conc}(\beta) \fatsemi \mathrm{seq}(\delta))) && \text{Expand } c \Downarrow (- \fatsemi -). \\
&= c \downarrow ((c \Downarrow \mathrm{conc}(\beta)) \bullet ((c \downarrow (c \Downarrow \mathrm{conc}(\beta))) \Downarrow \mathrm{seq}(\delta))) && \text{Expand two } c \Downarrow \mathrm{conc}(\beta). \\
&= c \downarrow (\beta(\llbracket q_\beta \rrbracket_c) \bullet ((c \downarrow \beta(\llbracket q_\beta \rrbracket_c)) \Downarrow \mathrm{seq}(\delta))) && \text{Expand } - \Downarrow \mathrm{seq}(\delta). \\
&= c \downarrow (\beta(\llbracket q_\beta \rrbracket_c) \bullet \delta(\mathrm{roll}_\delta(c \downarrow \beta(\llbracket q_\beta \rrbracket_c), \llbracket q_\delta \rrbracket_{c \downarrow \beta(\llbracket q_\beta \rrbracket_c)}))) && \text{Expand } c \downarrow (- \bullet -). \\
&= (c \downarrow \beta(\llbracket q_\beta \rrbracket_c)) \downarrow \delta(\mathrm{roll}_\delta(c \downarrow \beta(\llbracket q_\beta \rrbracket_c), \llbracket q_\delta \rrbracket_{c \downarrow \beta(\llbracket q_\beta \rrbracket_c)})) && \text{Substitute } c' \text{ for } c \downarrow \beta(\llbracket q_\beta \rrbracket_c). \\
&= c' \downarrow \delta(\mathrm{roll}_\delta(c', \llbracket q_\delta \rrbracket_{c'})) && \text{This } c' \text{ is that of Figure 12.}
\end{aligned}
$$

Fig. 13. We formalize the schedule discussed in Section 2.3 as z for the systems β and δ defined in Figure 11. We interpret z by performing $z(c)$, showing how execution proceeds from Figure 7 to Figure 12.

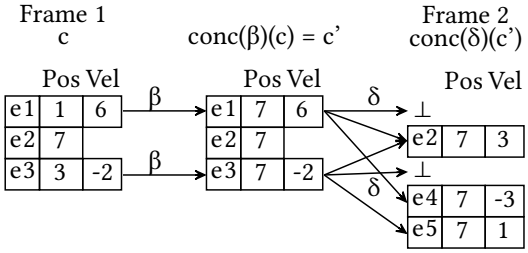
4.1 Concurrency model

Our characterization of concurrency in Core ECS rests on the question of whether one mask is visible during the production of another. Since masks represent state updates, this notion of visibility establishes an ordering relationship between updates. The design of schedules reflects this idea. Recall from Sections 3.5 and 3.6 that in the interpretation of a schedule to produce a composite mask, *sequential* means that the effects of some constituent mask are visible in subsequently produced masks (whether by the same system or others). Conversely, *concurrent* means that the effects of some constituent mask are **not** visible in subsequently produced masks.

Figure 14 illustrates the distinction between sequential and concurrent schedules with the application of two schedules to a state where two moving objects are about to converge on the same stationary object (Frame 1 in Figure 3). After $\mathrm{conc}(\beta)$ is applied to c, should both e_1 and e_3 collide with e_2? Even though both moving objects, paired with the stationary object, are in the entity matches of collision system δ for state c', under schedule $\mathrm{seq}(\delta)$ only one invocation of f_δ occurs because its effect is visible when the second is considered (Figure 14a). On the other hand, under schedule $\mathrm{conc}(\delta)$ both collisions occur, resulting in a lost write to the stationary object's velocity (Figure 14b). Were this a real parallel implementation, the question of which of the concurrent writes would complete last would be settled by scheduler non-determinism.



(a) Schedule $\mathrm{conc}(\beta) \fatsemi \mathrm{seq}(\delta)$ calls the collision system δ only once: After e_2 gets a velocity it is no longer a result of q_δ.



(b) Schedule $\mathrm{conc}(\beta) \fatsemi \mathrm{conc}(\delta)$ calls the collision system δ twice on e_2, causing a lost write (of -1) to its velocity.

Fig. 14. Comparison of schedules corresponding to the scenarios of Figures 3a and 3b.

We therefore analyze the concurrency of ECS programs by mimicking scheduler non-determinism, and adopt a relaxed perspective of the order in which masks are produced under concurrent schedules $\mathrm{conc}(-)$ and $(- \parallel -)$. For a given system s, while $\mathrm{seq}(s)$ remains beholden to the "consistent but unspecified total order" of entity matches (Section 3.3), the interpretation of $\mathrm{conc}(s)$ is regarded as taking place in an arbitrary order over entity matches (intra-system concurrency). Similarly, given sub-schedules z and z', under schedule $z \fatsemi z'$ the masks produced in z are produced prior to the masks produced by z', but the interpretation of $z \parallel z'$ is regarded to be the same as $z' \parallel z$ (inter-system concurrency).

To formalize this relaxed view, we characterize the admissible orderings of system function invocations under a schedule as the linearizations of a partial order. For schedule z at state c we define a partial order $\langle \mathbf{po}_c(z),\ \preceq_z \rangle$. Its elements are the system function invocations performed by $z(c)$, defined inductively.

$$
\begin{aligned}
\mathbf{po}_c(\mathrm{conc}(s)) &\triangleq \{(f_s, r) \mid r \in [\![q_s]\!]_c\} \\
\mathbf{po}_c(\mathrm{seq}(s)) &\triangleq \{(f_s, r) \mid r \in \mathrm{roll}_s(c, [\![q_s]\!]_c)\} \\
\mathbf{po}_c(z \parallel z') &\triangleq \mathbf{po}_c(z) \cup \mathbf{po}_c(z') \\
\mathbf{po}_c(z \fatsemi z') &\triangleq \mathbf{po}_c(z) \cup \mathbf{po}_{z(c)}(z')
\end{aligned}
$$

For brevity, we have treated the collection of entity matches $\{\vec{e}_i \mapsto \vec{w}_i\}_i$ as its graph $\{(\vec{e}_i, \vec{w}_i)\}_i$.

Next we give the ordering relation among system function invocations, inductively.

$$
\begin{aligned}
u \preceq_{\mathrm{conc}(s)} u' &\triangleq u = u' \\
(f_s, r) \preceq_{\mathrm{seq}(s)} (f_s, r') &\triangleq r \text{ and } r' \text{ are like-ordered in } \mathrm{roll}_s(c, [\![q_s]\!]_c) \\
u \preceq_{z \parallel z'} u' &\triangleq (u \preceq_z u') \vee (u \preceq_{z'} u') \\
u \preceq_{z \fatsemi z'} u' &\triangleq (u \preceq_z u') \vee (u \preceq_{z'} u') \vee (u \in \mathbf{po}_c(z) \wedge u' \in \mathbf{po}_{z(c)}(z'))
\end{aligned}
$$

For system s, schedule $\mathrm{conc}(s)$ imposes no dependencies among f_s invocations, whereas $\mathrm{seq}(s)$ orders f_s invocations as rolling the entity matches of s would ("like-ordered"), and vacuously does not order other invocations. Schedules formed by $(- \parallel -)$ require only that the ordering dependencies of the sub-schedules be respected, while schedules formed by $(- \fatsemi -)$ additionally require that one side applies wholly after the other (by recursing on an updated state).

Every linearization of $\mathbf{po}_c(z)$ thus respects $(- \preceq_z -)$ and fits our relaxed perspective of the order in which masks are produced. Furthermore, every such linearization corresponds to a composite mask, $f_1(r_1) \bullet \ldots \bullet f_n(r_n)$, and we will refer to the linearization and its composite mask interchangeably.

4.2 Deterministic concurrency

We can now identify a class of concurrent ECS programs which are nonetheless deterministic under scheduler non-determinism. Our result can be summarized imprecisely: A system that does not obtain entities from component values and has only a singleton query vector is deterministic under schedule $\mathrm{conc}(-)$, and two sub-schedules that write to disjoint sets of component labels are deterministic under schedule $(- \parallel -)$. Schedules $\mathrm{seq}(-)$ and $(- \fatsemi -)$ are always deterministic because they are not concurrent (they allow only one order of system function executions).

Our analysis is based on the observation that masks affecting disjoint locations in ECS state *commute*. Extending this observation, a schedule for which every concurrent pair of masks commutes is safe to execute concurrently. To formalize this argument, we will first identify the conditions under which a schedule gives rise to determinism, and next describe how to construct such schedules. To get started, we give a few definitions.

Definition 4.1 (Mask equivalence). Masks m and m' are *equivalent* (written $m \simeq m'$) if for all states c we can arrange that $c \downarrow m = c \downarrow m'$ (i.e., by a correlated choice of fresh entities).[7] Similarly, two linearizations of a schedule are equivalent if their corresponding masks are.

Definition 4.2 (Mask commutativity). Masks m and m' are said to *commute* if $m \bullet m' \simeq m' \bullet m$.

Definition 4.3 (Schedule safety). A schedule z is said to be *safe at state c* if for every concurrent pair (f_s, r) and $(f_{s'}, r')$ in $\mathbf{po}_c(z)$ the masks $f_s(r)$ and $f_{s'}(r')$ commute.

Definition 4.4 (Schedule determinism). A schedule z is said to be *deterministic at state c* if all linearizations of $\mathbf{po}_c(z)$ are equivalent.

Theorem 4.5 (Schedule safety implies schedule determinism). *Any schedule z safe at state c is deterministic at state c.*

Proof. Since we may transform one linearization into any other by a chain of order-respecting adjacent transpositions [Etienne 1984], it suffices to show that any two linearizations related by a single order-respecting adjacent transposition are equivalent. Given m and m' differing only in the order of two adjacent masks:

$$m = f_1(r_1) \bullet \ldots \bullet f_a(r_a) \bullet f_b(r_b) \bullet \ldots \bullet f_n(r_n)$$
$$m' = f_1(r_1) \bullet \ldots \bullet f_b(r_b) \bullet f_a(r_a) \bullet \ldots \bullet f_n(r_n)$$

Since both m and m' respect $\mathbf{po}_c(z)$ but are different, it must be that (f_a, r_a) and (f_b, r_b) are concurrent. Given that and the assumption that z is safe at state c, we know that $f_a(r_a)$ and $f_b(r_b)$ commute. Therefore, $m \simeq m'$. □

Since any safe schedule is deterministic by Theorem 4.5, we now turn to a discussion of how to construct safe schedules. We will require a mechanism to examine which entities and components are affected by a schedule at some state in order to show that it is safe. For this we define the *influence* of a mask m, $\mathrm{infl}(m)$, or of a schedule z at some state c, $\mathrm{infl}_c(z)$, to be the subset of $E \times I$ pairs necessarily affected by it. In the case of a mask, the complement of infl(m) is the set of entities and components (as indexes into state) for which the state is unchanged, i.e. $(c \downarrow m)_i(e) = c_i(e)$ holds for all c. Influence is defined inductively for both masks (left) and schedules (right):

$$\begin{aligned}
\mathrm{infl}(\mathrm{attach}_i(e, k_i)) &\triangleq \{(e, i)\} & \mathrm{infl}_c(\mathrm{conc}(s)) &\triangleq \bigcup_{r \in [\![q_s]\!]_c} \mathrm{infl}(f_s(r)) \\
\mathrm{infl}(\mathrm{detach}_i(e)) &\triangleq \{(e, i)\} & \mathrm{infl}_c(\mathrm{seq}(s)) &\triangleq \bigcup_{r \in \mathrm{roll}_s(c, [\![q_s]\!]_c)} \mathrm{infl}(f_s(r)) \\
\mathrm{infl}(\mathrm{m} \bullet \mathrm{m}') &\triangleq \mathrm{infl}(m) \cup \mathrm{infl}(m') & \mathrm{infl}_c(z \parallel z') &\triangleq \mathrm{infl}_c(z) \cup \mathrm{infl}_c(z') \\
\mathrm{infl}(\nu(f)) &\triangleq \bigcap_{e \in E} \mathrm{infl}(f(e)) & \mathrm{infl}_c(z \fatsemi z') &\triangleq \mathrm{infl}_c(z) \cup \mathrm{infl}_{z(c)}(z') \\
\mathrm{infl}(\mathrm{nil}) &\triangleq \emptyset
\end{aligned}$$

The influence of a mask (or a schedule) is the union of those entity and component label pairs that it affects — a "memory footprint" — except in the case of $\nu(-)$. Since the interpretation $c \downarrow \nu(f)$ calls f on an fresh entity not present in c, the influence on that entity and its components is not visible to any concurrent mask, so we discard that influence using intersection. Of the influence generated by f, only that invariant to the choice of fresh entity is preserved.

With influence we can now make statements about when masks or schedules commute, which will enable us to state rules of safe construction. We do so in the following proofs:

[7] Since the choice of a fresh entity to satisfy $\nu(-)$ is arbitrary, interpreting $c \downarrow \nu(\lambda\, e.\ \mathrm{attach}_i(e, k_i))$ may produce distinct result states. However, since E is an opaque parameter to Core ECS, those states are indistinguishable.

Lemma 4.6. *For masks m and m', if* $\mathrm{infl}(m) \cap \mathrm{infl}(m') = \emptyset$ *then m and m' commute.*

Proof. For masks not involving $\nu(-)$, this is immediate: such masks are compositions of attach, detach, and nil, which will commute because they affect distinct entity-component indices.

For masks involving $\nu(-)$ we arrange for disjoint fresh entities to be provided. Then the masks will behave as though those provided entities were inlined within them. They reduce to masks not involving $\nu(-)$ whose influences remain disjoint, and commute as described above. □

Theorem 4.7. *Construction of safe schedules by cases is as follows:*

1. *The schedule* $\mathrm{seq}(s)$ *is safe at all states for any system s.*
2. *The schedule $z \fatsemi z'$ is safe at state c if z is safe at c and z' is safe at $z(c)$.*
3. *The schedule* $\mathrm{conc}(s)$ *is safe at state c if* $\mathrm{infl}(f_s(r)) \cap \mathrm{infl}(f_s(r')) = \emptyset$ *for every distinct pair of entity matches r and r' in $[\![q_s]\!]_c$.*
4. *The schedule $z \parallel z'$ is safe at state c if z and z' are safe at c and* $\mathrm{infl}_c(z) \cap \mathrm{infl}_c(z') = \emptyset$.

Proof. By cases:

1. Since $\mathrm{seq}(s)$ has no concurrent pairs of system function invocations, it is vacuously safe.
2. Any two concurrent invocations in $z \fatsemi z'$ are either both in z or both in z', which are safe at their respective states, and since there are no other pairs to consider, $z \fatsemi z'$ is safe at c.
3. Since the pairs of concurrent invocations in $\mathrm{conc}(s)$ yield pairs of masks that satisfy Lemma 4.6, $\mathrm{conc}(s)$ is safe at c.
4. Any two concurrent invocations in $z \parallel z'$ that are both in z or both in z' are safe because z and z' are safe at c. For pairs of concurrent invocations that straddle z and z', since the influences of z and z' are disjoint, every pair of invocations yields a pair of masks that satisfy Lemma 4.6. Therefore $z \parallel z'$ is safe at c. □

Any schedule constructed by the rules in Theorem 4.7 is safe, and therefore by Theorem 4.5 is deterministic. However, in practice, the full force of Theorem 4.7 requires understanding the influence of each system under all possible runtime inputs, which can be a nontrivial analysis. We close this section with two corollaries of Theorem 4.7 that can be judged statically.

Corollary 4.8. *If schedules z and z' are safe and influence disjoint sets of component labels, then $z \parallel z'$ is safe (and thus deterministic) on any state c.*

Corollary 4.9. *For a system s, if $q_s = \langle q_1 \rangle$ is a singleton-vector and the component types of $(\!|q_1|\!)$ do not reference the entity type E, then* $\mathrm{conc}(s)$ *is safe (and thus deterministic) on any state c.*

Corollary 4.9 says a system that only ever influences one entity per invocation, run concurrently with itself, won't influence the same entity in two of the invocations. By restricting the components, no entity enters a system invocation via component values. By restricting the query vector to one query, only one entity enters each system invocation. This is possible to judge statically by examining the the fixed query vector for a system and the component types in the schema K. Use of a system in this way is a parallel map!

Meanwhile Corollary 4.8 says that if two sub-schedules work over entirely disjoint sets of component labels, then it doesn't matter whether they influence the same entities — their influences must necessarily be disjoint. A benefit of the ECS pattern is that an existing ECS program can be easily extended with new systems and components in response to design changes, and so this scenario arises frequently in practice. Indeed, it is recognizably the act of running distinct tasks on distinct regions of memory.

It may seem quite obvious that if concurrent tasks are restricted to only modify disjoint regions of memory, we will have deterministic concurrency; however, as we find in Section 5, none of the ECS

implementations we studied fully implement even this degree of concurrency. Our results establish a kind of "green light", go-ahead signal for concurrency in ECS: there is no theoretical obstacle to achieving this degree of concurrency, only practical obstacles dependent on implementation decisions.

5 ECS IN PRACTICE

In this section we survey five prominent open-source ECS frameworks listed in Table 1, reflecting on Core ECS for comparison. We focused on ECS frameworks with open-source implementations that prioritize efficient execution, and biased our selection toward widely-used frameworks. In Section 5.1, we discuss the five frameworks' approaches to two key implementation decisions: their approach to fresh entity generation, and their choice of component storage strategy.

With these implementation decisions in view, Section 5.2 turns to a comparison of the frameworks' support for concurrency. Our overall finding is that, compared to Core ECS, all five of the frameworks we surveyed *disallow* a subset of the safe schedules that Theorem 4.7 identifies. That is, in all five frameworks there is a gap between the degree of deterministic concurrency that the ECS pattern (as modeled by Core ECS) *can* support, and the degree of deterministic concurrency that ECS frameworks *do* support. In particular, none of the five frameworks support the ability to attach a new component to an entity concurrently with another mutation.

The five frameworks we survey (in order of popularity[8]) are: *Bevy ECS* [Anderson and Bevy Contributors 2024], a Rust ECS framework developed for the Bevy game engine; *EnTT* [Caini 2024d], an ECS framework for C++, known for its use in Minecraft [Mojang AB and Microsoft Corporation 2024]; *Flecs* [Mertens 2024a], an ECS framework for C and C++; *Specs* [Schaller 2023], a Rust ECS framework developed for the Amethyst game engine [Kalderon 2021]; and *apecs* [Carpay 2018a], an ECS framework for Haskell. These frameworks variously bill themselves as "fast" [Carpay 2018a] (or "[i]ncredibly fast" [Caini 2024d]), "massively parallel" [Anderson and Bevy Contributors 2024], supporting multithreading with a "fast lockless scheduler" [Mertens 2024a], and offering "easy parallelism" and "high performance" [Schaller 2023] – claims that existing ECS benchmarking efforts have sought to validate [Schmierer 2017; Beimler 2024]. Our purpose in this section, however, is not a quantitative performance assessment, but rather a qualitative comparison of design and implementation decisions made in the interest of performance.

5.1 Implementation techniques

We guide our exploration of implementation techniques with two questions that often drive the design of an ECS framework. How are fresh entity identifiers generated? What strategy is used to store components? (One further question, how a programmer specifies the system schedule, is explored by Section 5.2.) The way in which an implementation answers these questions will directly influence the degree of concurrency (and coordination) in that implementation. In practice, the answers to these questions tend to be correlated, as (for instance) certain strategies for storage implicate certain strategies for indexing that storage with entity identifiers. Moreover, the desire to minimize indirection, speeding up iteration over (and updates to) the entity-component association, is in tension with the desire to maximize efficient querying and structure changes, aspects which may benefit from indirection.

[8] As indicated by GitHub stars, measured 2024 October 9.

[9] "In general, the entire registry isn't thread safe as it is. Thread safety isn't something that users should want out of the box for several reasons. Just to mention one of them: performance." [Caini 2024a]

[10] Apecs removed concurrency support 19 versions and 5 years ago with the update from v0.4.1.1 to v0.5.0.0 [Carpay 2018c,b]. The removed functions `pmap` and `concurrency` provided intra- and inter- system concurrency, respectively. "Provides zero protection against race conditions and other hazards, so use with caution." [Carpay 2018c]

Framework	Stars	Language	Entity generation	Strategy	Scheduling
Bevy ECS	35,6xx	Rust	Generational indexing	Archetype	Semi-automated
EnTT	10,1xx	C++	Generational indexing	Columnar	Manual[9]
Flecs	6,4xx	C	Generational indexing	Archetype	Semi-automated
Specs	2,5xx	Rust	Generational indexing	Columnar	Semi-automated
apecs	392	Haskell	Sequential numeric	Columnar	Manual[10]

Table 1. Summary of the ECS frameworks we surveyed. The "Stars" column is a count of GitHub stars. "Entity generation" refers to frameworks' fresh entity generation approach, as discussed in Section 5.1. "Strategy" refers to frameworks' storage strategy, as discussed in Section 5.1. "Scheduling" refers to frameworks' system scheduling interface, as discussed in Section 5.2.

Fresh entity generation. As discussed in Section 3, to use the ECS pattern in some language, a programmer designates a type to represent entity identifiers, and implements some means to generate fresh elements of that type. There are a few obvious implementation-level approaches to this, such as the use of *sequential numeric identifiers*, or generating *random numbers* with many bits. While these approaches are not widely used in practice, it is helpful to think through their trade-offs before we consider the more commonly used technique, called *generational indexing*. What is the state required to generate entities? What coordination is necessary to ensure that concurrently generated entities are unique?

Using sequential numerical identifiers to generate entities, the approach employed by apecs [Carpay 2024b], is reminiscent of an auto-incrementing primary key in a database table. A fresh entity identifier is obtained by incrementing a shared global variable. This strategy requires a minimum of state and only modest coordination when generating new entities: A multithreaded framework may use an atomic integer to generate distinct entities concurrently.

Random identifiers require no state nor coordination to ensure that concurrently generated entities are unique. A fresh entity identifier is obtained by generating a random number with enough bits of entropy to minimize the possibility of a collision. This approach eliminates state and coordination at the expense of necessitating a larger identifier size.

Generational indexing is a more sophisticated strategy in which an entity identifier consists of an index part and a generation part [West 2018; Sardois 2021]. Flecs, Specs, EnTT, and Bevy ECS all generate fresh entities using generational indexing [Mertens 2020b; Schaller 2023; Caini 2024b; Bevy Contributors 2024c]. In this approach, a fresh entity identifier is either produced by recycling a previously used index or minting a new one. This strategy requires more state and coordination than the other strategies we mention, but produces entity identifiers from which an array index may be extracted: Array indexes generated this way help to reuse the allocated space in arrays (as compared with sequential numerical identifiers) by reducing the frequency of array re-allocations that require global coordination. As such, generational indexing presents another trade off, moving coordination from those arrays (component storage) to fresh entity generation.

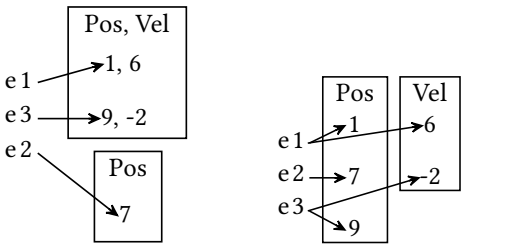
To emphasize the need for coordination, we briefly describe (a simplified form of) the generational indexing strategy used by Schaller [2023]. State includes a cache of currently unused indexes, a registry of the latest generation value per index, and a high-water mark for all indexes. To determine whether a given entity is live, retrieve the entity index's latest generation value from the registry; the entity is live if its generation value matches that of the registry. Accordingly, to mark an index as no longer live, increment its latest generation value in the registry.[11] To make an index available for recycling, as part of marking it no longer live, add it to the cache of unused indexes. Finally, to

[11] Schaller [2023] instead marks the generation as "dead" by changing which contstructor wraps the generation value.

obtain a fresh entity by recycling an index or minting a new one: Recycle a previously used index by removing it from the cache of unused indexes and pairing it with its latest generation value in the registry[12]; since the entity generated this way is the only instance of its index with that generation value, it is distinguishable from previous incarnations. To mint a new index, increment the global high-water mark of indexes and pair it with some initial generation value. Performing all of these operations concurrently requires some amount of global coordination.

Component storage strategies. We observe two main strategies for the storage of components, calling one *columnar* and the other *archetype* (as seen in Gillen [2021]).

Archetype-style frameworks logically group entities by the set of component labels that they have, and place the component values for each logical group (each *archetype*) in the same region of memory. For example, in Figure 15a, (**Pos**, **Vel**) is an archetype for entities with both a **Pos** and a **Vel** component. Flecs and Bevy ECS are archetype-style frameworks [Mertens 2020a; Bevy Contributors 2024b]. This strategy facilitates the execution of queries, because examination of the component labels in each archetype immediately establishes the groups of entities that match the query. This strategy may also facilitate cache locality in a tight loop over those entities, if the component values for each entity are stored together. However, the cost of moving component data between regions when an entity's set of components changes may be a disadvantage. A framework may attempt to intelligently deploy distinct layouts for different archetype regions, based on the number of entities and the size of their component data.



(a) Archetypes are ad hoc regions filled with entities having like-components.

(b) Columnar frameworks maintain components separately by label.

Fig. 15. The two main component storage strategies.

Columnar-style frameworks group component values with the same component label together, enabling distinct storage styles for distinct component labels (Figure 15b). EnTT, Specs, and apecs are columnar-style frameworks [Caini 2024c; Schaller 2023; Carpay 2024a]. This strategy minimizes the cost of updating the entity-component association by requiring changes in only one logical column; however, it may suffer from poor locality when accessing many component values of a single entity in a tight loop over entities. There are many concrete column storage styles and we list only a few here for flavor.

- Array — By obtaining an index from an entity identifier, component data can be stored in an array. A sentinel value or bit-mask may indicate which indexes are in use (i.e. which entities have the given component). Unused indexes contribute to fragmentary memory use. Paired with generational indexing, array reallocation and fragmentation can be minimized.
- Dense array — Adding indirection between entity identifiers and component array indexes (e.g. using a sparse-set structure) may alleviate the problem of fragmentary memory use and reduce the preference for generational indexing.
- Map — Choosing an off-the-shelf hash-map or tree-map abstracts the problem of storage completely from the strategy for indexing, at potential cost of iteration speed.

The many trade-offs between component storage are also influenced by the use cases of different applications, and thus extend beyond the scope of our purpose here.

[12] Schaller [2023] "raises" the "dead" generation and increments it.

5.2 Practically available concurrency

To explore the concurrency available in the ECS frameworks of Table 1, we compare the expressiveness of system scheduling in those frameworks with that of Core ECS schedules (Section 3.6). We find that there are broadly two approaches to system scheduling interfaces in the ECS frameworks we examine: *manual* and *semi-automated*. Our examples in this section will use the inertia (β) and collision (δ) systems from our toy physics simulation example in Sections 2 and 3, and for demonstration purposes we throw in two more systems: a rotation system (which we call γ) and a render system (which we call η).

For frameworks in the manual camp, there is no first-class concept of scheduling systems. Instead a programmer schedules systems directly by invoking them on the ECS state in the desired order, typically in the context of an outer "main loop". Each system function expresses a query over the ECS state, and iterates sequentially over entity matches, using facilities provided either by the ECS framework or by the host language. Among the frameworks we surveyed, this is the approach taken by apecs [Carpay 2019] (which removed its support for concurrency) and EnTT [Kernick 2020] (for which documentation implies that concurrency is not an intended use case). Such lightweight scheduling may be straightforward and incur little overhead, but it places the burden of scheduling, and thus of correctly managing concurrency (if any), on the programmer. In practice, this amounts to sequential schedules composed of $\mathrm{seq}(-)$ and $(- \fatsemi -)$, as in the following Core ECS schedule at left, and corresponding hypothetical EnTT schedule at right.

$$\mathrm{conc}(\beta) \fatsemi \mathrm{seq}(\gamma) \fatsemi \mathrm{seq}(\delta) \fatsemi \mathrm{seq}(\eta)$$

```
inertiaSystem(state);
rotationSystem(state);
collisionSystem(state);
renderSystem(state);
```

For frameworks in the semi-automated camp, one commonality is that calls to system functions may be managed by a framework main loop that the programmer configures to achieve inter-system concurrency. Flecs, Specs, and Bevy ECS (which all advertise their support for "parallelism") offer this kind of semi-automated scheduling. Such schedules may consist of a sequential spine of $(- \fatsemi -)$ that orders concurrent groups of systems composed with $(- \parallel -)$. The choice of sequential or concurrent execution of individual systems is usually internalized into the implementation of those systems, rather than being made explicit when they are scheduled. We evoke this pattern with the following Core ECS schedule at left, and corresponding hypothetical Bevy ECS schedule at right.

$$\begin{aligned}&(\mathrm{conc}(\beta) \parallel \mathrm{seq}(\gamma)) \fatsemi \\ &\mathrm{seq}(\delta) \fatsemi \\ &\mathrm{seq}(\eta)\end{aligned}$$

```
let mut z = Schedule::default();
z.add_systems((
    (inertia, rotation),
    collision,
    render
  ).chain());
```

Intra-system concurrency, as in $\mathrm{conc}(\beta)$, is possible to achieve via parallel iteration. The most common idiom is a framework-provided parallel-map function that takes a function to call once per entity match, just as in Core ECS. We show an example of this idiom as it appears in Bevy ECS in Figure 16. This idiom also highlights a difference between Core ECS and the frameworks we investigated: Core ECS relocates intra-system concurrency from the system to the schedule.

Continuing with the Bevy ECS inertia system example in Figure 16, we observe that the semi-automated scheduling regimes of Flecs, Specs, and Bevy ECS are all designed to avoid *write conflicts* between concurrent systems. For the two Rust frameworks, Bevy ECS and Specs, this avoidance is achieved by shallowly embedding the problem into the borrow checker: In Figure 16 the position

```
fn inertia(mut q: Query<(&mut Pos, &Vel)>) {
  q.par_iter_mut().for_each(|(mut p, Vel(v))| { p.0 += v; });
}
```

Fig. 16. A Bevy ECS inertia system, expressed as a Rust function, that demonstrates intra-system concurrency (as in conc(β)) and exclusive access to (logical) stores.

store is mutably borrowed by the inertia system in `&mut Pos`, and the velocity store is immutably borrowed by `&Vel`. No concurrent system may borrow the position store, and no concurrent system may mutably borrow the velocity store. The documentation for Bevy ECS explicitly addresses this restriction: "Not all systems can run together: if a system mutably accesses data, no other system that reads or writes that data can be run at the same time. These systems are said to be **incompatible**." [Bevy Contributors 2024a]

Furthermore, in all three of the multithreaded frameworks we investigated, it is not possible to attach a new component to an entity in any concurrent setting. The problem may lie in the need to potentially allocate storage for the new component value. Flecs documentation provides us with a name for this problem: a *structure change* is altering the set of components attached to an entity. In the Flecs authors' words, "By default systems are ran while the world is in 'readonly' mode, where all ECS operations are enqueued as commands. Readonly here means that structural changes, such as changing the components of an entity are deferred." [Mertens 2024b]

All three of the multithreaded ECS frameworks in Table 1 provide the same two workarounds for these restrictions imposed to avoid write conflicts and concurrent structure changes.

1. Deferring modifications to entities, by storing them in a buffer to be applied sequentially by the ECS framework later (called *lazy updates* by Specs [Schaller 2022] and *parallel commands* by Bevy ECS [Bevy Contributors 2024d]) allows any modification to be expressed in a concurrent context.
2. Running a system in a single thread with exclusive access to ECS state (called an *immediate system* in Flecs and an *exclusive system* in Bevy ECS) allows it to make any modifications in any component store.

Of these, only deferred modifications (1) occur in concurrent contexts, and so we ignore exclusive systems (2) in further discussion.

The well-intentioned attempts by the authors of these frameworks to provide concurrency free of data races results in a zoo of mutation categories, which we catalog in Table 2. With each mutation category in the catalog we provide an example Core ECS system which, if translated to Flecs, Specs, or Bevy ECS, would demonstrate a distinct variety of mutation in that framework (though some categories do not have a translation to some frameworks). As in Figure 16, a component that appears in the query vector of these example systems can be regarded as owned (or borrowed mutably) in a translation. We ask the reader to consider whether and how each example may be translated.

In the multithreaded ECS frameworks that we studied, only the first mutation category in Table 2 (Owned update) is possible to execute concurrently. The four "deferred" mutation categories are nominally "parallel" according to the frameworks' documentation, but factually they are serialized. The remaining three "owned" mutation categories (owned insert, owned initialize, and owned delete) are, in concurrent contexts, only possible to express as the corresponding deferred structure changes. These owned structure changes may cause allocation in a component store (insert, initialize) or in fresh entity structures (initialize); of them one should be a local operation for columnar frameworks (delete) such as Specs.

Mutation Category	Example System	Side condition
Owned update	$\langle \text{incl}_{\mathbf{Pos}} \rangle,\ \lambda(e,\ p).\ \text{attach}_{\mathbf{Pos}}(e, 0)$	
Owned insert	$\langle \text{excl}_{\mathbf{Pos}} \rangle,\ \lambda(e,\ -).\ \text{attach}_{\mathbf{Pos}}(e, 0)$	
Owned initialize	$\langle \text{anyway}_{\mathbf{Pos}} \rangle,\ \lambda(e,\ p).\ \nu(\lambda e'.\ \text{attach}_{\mathbf{Pos}}(e', 0))$	
Owned delete	$\langle \text{incl}_{\mathbf{Pos}} \rangle,\ \lambda(e,\ p).\ \text{detach}_{\mathbf{Pos}}(e)$	
Deferred update	$\langle \text{anyway}_{\mathbf{Pos}} \rangle,\ \lambda(e,\ p).\ \text{attach}_{\mathbf{Vel}}(e, 1)$	e has **Vel**
Deferred insert	$\langle \text{anyway}_{\mathbf{Pos}} \rangle,\ \lambda(e,\ p).\ \text{attach}_{\mathbf{Vel}}(e, 1)$	e does not have **Vel**
Deferred initialize	$\langle \text{anyway}_{\mathbf{Pos}} \rangle,\ \lambda(e,\ p).\ \nu(\lambda e'.\ \text{attach}_{\mathbf{Vel}}(e', 1))$	
Deferred delete	$\langle \text{anyway}_{\mathbf{Pos}} \rangle,\ \lambda(e,\ p).\ \text{detach}_{\mathbf{Vel}}(e)$	e has **Vel**

Table 2. Categories of mutations having implications for concurrent execution in the multithreaded ECS frameworks we studied. Those that would influence the same component store that they query are regarded as owning that store. When a store is not owned, influence against it must be deferred. Insertions and deletions are both structure changes which, respectively, potentially allocate or are assumed to not allocate.

If the premises of safe schedule construction in Theorem 4.7 are observed, any of the mutation categories in Table 2 are safe to execute concurrently, and will produce deterministic behavior. In principle, it would seem that frameworks would let an ECS program express any safe schedule. Corollaries 4.8 and 4.9 identify two subsets of safe schedules that can be checked statically. It is clear that a structure change involving one component, executed concurrently with mutations at different components, is safe and deterministic because — as in the frame rule in a concurrent separation logic — these actions deal with disjoint regions of memory. Yet none of the multithreaded ECS frameworks we investigate fully support Corollary 4.8. These practical frameworks disallow obviously correct forms of concurrency, expressing a bias toward domains where the structure of entities does not change frequently.

6 RELATED WORK

Join calculus. The join calculus of Fournet and Gonthier [1996, 2000] is a combination of a minimal ML-inspired language with "processes" that may be run concurrently with other processes. Running a process P concurrently with another Q is written with a parallel composition operator as in $P \mid Q$, similar to our notion of concurrent composition of Core ECS schedules. Processes are not necessarily required to return a value and hence are not ML expressions, much like Core ECS schedules do not return a value and are not expressions in the underlying programming language. Via a syntax sugar over **let** expressions, a side-effecting ML expression E may be sequenced with either an expression or a process P, as in $E\ ;\ P$, which is similar to our notion of sequential composition of schedules, except that it interlaces ML expressions with processes.

Indeed, the join calculus allows process abstractions to be defined and run within expressions, and functions containing ML expressions to be defined and run within processes. This sort of interlacing of the two domains is unlike Core ECS, which maintains a schedule at the top level, with expressions of the underlying language appearing only at the leaves within systems.

Furthermore, processes may be called from ML expressions and return results to ML expressions, communicating across these boundaries via "channels". The join calculus additionally defines a variety of "pattern-matching" "inter-process synchronization" primitives. These are unlike Core ECS in that schedules neither receive input from expressions, nor return results, nor synchronize with each other — all communication between systems in Core ECS is in the form of updates to the single entity-component association.

Relational database management systems. It is commonly observed that the ECS pattern is in effect a very narrow and regimented use of a relational database management system (RDBMS) [Bilas 2002; Martin 2007b; Mertens 2023; Gutekanst 2022; Borisova 2024]. The entity-component association can be encoded directly with a single RDBMS table using the "entity attribute" schema. A columnar layout can be encoded by storing each component type in a separate single-column table with a foreign key that references an entity table. Through use of dynamically created multi-column tables, an archetype layout can also be encoded. Given any such layout, ECS systems are possible to encode with a select query followed by an update query, combined or with an embedded third-party programming language in between , all persisted via stored procedures. Finally, triggers could be set up to activate the stored procedures based on their select queries.

Despite these subsuming similarities, the historical development of the ECS pattern, on underpowered hardware in which all memory is allocated to the ECS program, emphasizes that it is distinct from RDBMS [Bilas 2002; West 2018]. Historically and presently, ECS frameworks are designed for efficiency first, and this focus leads to a few differences from RDBMSs: The ECS pattern is not concerned with persistent storage. It is common for the entity-component association to be represented plainly as a struct of arrays [West 2018; Mertens 2024a; Schaller 2023]. Furthermore, the ECS pattern focuses on a small schedule of queries that are fixed during runtime, and so sidesteps the whole question of query optimization characteristic of RDBMSes.

Deterministic concurrent programming models. There is a very long tradition of work on abstractions for deterministic concurrent programming [Tesler and Enea 1968; Kahn 1974; Arvind et al. 1989; Peyton Jones et al. 2008; Bocchino et al. 2009; Prokopec et al. 2012; Kuper and Newton 2013]. In general, deterministic concurrent programming models must somehow restrict access to mutable shared state. Our determinism result in Section 4.2 depends on the fact that concurrent tasks only access *disjoint* state. This approach to deterministic concurrency is similar to that taken by Deterministic Parallel Java [Bocchino et al. 2009, 2011]. We are not aware of any other work on determinism of ECS programs specifically, but given the disjointness condition, our determinism result is conceptually straightforward. Abstractions such as LVars [Kuper and Newton 2013; Kuper et al. 2014], on the other hand, do allow concurrent tasks to access overlapping state in arbitrary order, but retain determinism by carefully restricting *how* the state can be updated and queried. In future work, it would be interesting to investigate how the ECS programming model could be combined with the approach taken by deterministic concurrency abstractions that allow some degree of (well-behaved) overlap in the memory footprint of tasks, rather than the total disjointness that we currently consider.

7 CONCLUSION AND FUTURE WORK

We have presented Core ECS, a formal model for the ECS software design pattern, that abstracts away from the implementation details of specific ECS frameworks to reveal what we believe are the essential characteristics of the ECS pattern. We precisely characterized concurrency in the ECS pattern using Core ECS as its model, we provided rules of construction for well-behaved concurrent ECS programs, and proved that those programs are deterministic — invocations of systems that mutate disjoint state are safe to execute concurrently. Our determinism result suggests that the ECS pattern can be viewed as a general deterministic concurrent programming model. While it is unsurprising that concurrent mutations of disjoint state give rise to determinism, through our result we identified that only a small part of the deterministic concurrency available in the ECS pattern is also available in commonly used ECS frameworks. By identifying where ECS frameworks restrict concurrency unnecessarily, our result can guide the design of new ECS frameworks that

do not require such restrictions, opening up opportunities for efficient and correct concurrent programming.

There are several possible directions for future work; here, we highlight two of them:

Quantitative evaluation of alternative parallel ECS implementation techniques. Through our comparison of deterministic concurrency in Core ECS with that available in multithreaded ECS frameworks (Section 5.2), we identified forms of concurrency not served by the common existing implementation techniques for the ECS pattern. While it is notable that frameworks' avoidance of write conflicts prevents two concurrent systems from writing to the same component store at disjoint sets of entities, we find it more surprising that the avoidance of concurrent structure changes prevents two concurrent systems from adding distinct components to entities, or from creating new entities with any components. We are therefore very interested to investigate alternative implementation techniques that rely less on global structures (such as those required for generational indexing), rely less on correlated data (such as bitmasks that are often paired with arrays in columnar component storage), and in general avoid mutual exclusion.

We believe a significant runtime advantage may be available via a technique that eschews fanatical prioritization of throughput, and instead amortizes costs, reduces contention, and consequently maximizes available concurrency. The ECS pattern has the potential to be a new concurrent programming model or even a compiler target. We propose to implement several existing and new techniques and measure their wall-clock time and relative speedup with a suite of benchmarks more broad than traditional simulations lacking in structure changes.

Query expressivity. The querying capabilities we describe in Section 3.2 are intentionally minimal, to better focus on the essence of the ECS pattern. However, this minimalism is quite limiting along a few distinct dimensions. Most obviously, taking the cartesian product of n queries will result in $O(x^n)$ system invocations, which scales poorly with an increasing number of queries. In many cases a system may only do useful work on a very small number of entity matches, such as in a kinematics simulation [Barnes and Hut 1986] that only concerns points that are sufficiently close. Extending Core ECS with a more expressive query langauge, such as non-cartesian joins or filters over individual queries, may alleviate this issue. Alternatively, allowing a system to perform a series of queries, each informed by the previous, may provide a completely different solution.

ACKNOWLEDGMENTS

To you, for reading this.

8 DATA-AVAILABILITY STATEMENT

There is no distinct artifact for this work. Our contributions are the design of a formalism to represent the ECS pattern (Core ECS), and the conclusions we have drawn by analyzing existing frameworks through it. While we used Haskell to implement the formalism twice during the design process, those implementations are changed slightly through the process of shallowly embedding in the language. Therefore we do not present those implementations as an artifact at this time.

REFERENCES

Carter Anderson and Bevy Contributors. 2024. Bevy Engine. https://bevyengine.org/. Releases https://docs.rs/bevy_ecs/latest/bevy_ecs/. Accessed 2024-10-09.

Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. 1989. I-structures: data structures for parallel computing. *ACM Trans. Program. Lang. Syst.* 11, 4 (Oct. 1989).

Josh Barnes and Piet Hut. 1986. A hierarchical O(N log N) force-calculation algorithm. *Nature* 324, 6096 (1986), 446–449.

Alex Beimler. 2024. Entity-Component-System Benchmarks. https://github.com/abeimler/ecs_benchmark. Accessed 2024-10-9.

Bevy Contributors. 2024a. Bevy Engine. Documentation https://docs.rs/bevy_ecs/latest/bevy_ecs/system/index.html#system-ordering. Accessed 2024-10-16.

Bevy Contributors. 2024b. bevy_ecs::archetype - Rust. Documentation https://docs.rs/bevy_ecs/latest/bevy_ecs/archetype/index.html. Accessed 2024-10-15.

Bevy Contributors. 2024c. Entity in bevy_ecs::entity - Rust. Documentation https://docs.rs/bevy_ecs/latest/bevy_ecs/entity/struct.Entity.html. Accessed 2024-10-15.

Bevy Contributors. 2024d. ParallelCommands in bevy_ecs::system. Documentation https://docs.rs/bevy_ecs/0.14.2/bevy_ecs/system/struct.ParallelCommands.html. Accessed 2024-10-16.

Scott Bilas. 2002. A Data-Driven Game Object System. Presentation https://www.gamedevs.org/uploads/data-driven-game-object-system.pdf. Video https://www.youtube.com/watch?v=Eb4-0M2a9xE. Audio https://www.gdcvault.com/play/1022543/A-Data-Driven-Object. Accessed 2024-04-04.

Robert L. Bocchino, Jr. et al. 2011. Safe nondeterminism in a deterministic-by-default parallel language. In *POPL*.

Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A type and effect system for deterministic parallel Java. In *OOPSLA*.

Ida Borisova. 2024. Unofficial Bevy Cheat Book – Intro: Your Data. https://bevy-cheatbook.github.io/programming/intro-data.html. Living https://github.com/bevy-cheatbook/bevy-cheatbook/blob/main/src/programming/intro-data.md. Accessed 2024-10-15.

Michele Caini. 2024a. Crash Course: entity component system. Heading "Multithreading". https://github.com/skypjack/entt/wiki/Crash-Course:-entity-component-system/aa053854f18d4589fe7ce6284752dfdbe1d5d3b0#multithreading. Accessed 2024-10-09.

Michele Caini. 2024b. Crash Course: entity component system. Heading "The Registry, the Entity and the Component". https://github.com/skypjack/entt/wiki/Crash-Course:-entity-component-system/aa053854f18d4589fe7ce6284752dfdbe1d5d3b0#the-registry-the-entity-and-the-component. Accessed 2024-10-09.

Michele Caini. 2024c. Crash Course: entity component system. Heading "All or nothing". https://github.com/skypjack/entt/wiki/Crash-Course:-entity-component-system/465d90e0f5961adc460cd9d1e9358370987fbcd3#all-or-nothing. Accessed 2024-10-15.

Michele Caini. 2024d. EnTT: Gaming meets Modern C++. https://skypjack.github.io/entt/. Living https://github.com/skypjack/entt. Accessed 2024-10-15.

Jonas Carpay. 2018a. Apecs: A Type-Driven Entity-Component-System Framework. Preprint https://github.com/jonascarpay/apecs/blob/master/apecs/prepub.pdf. Accessed 2024-10-09.

Jonas Carpay. 2018b. apecs: Fast ECS framework for game programming (v0.5.0.0). Release https://hackage.haskell.org/package/apecs-0.5.0.0. Accessed 2024-10-09.

Jonas Carpay. 2018c. Apecs.Concurrent (v.0.4.1.1). Module https://hackage.haskell.org/package/apecs-0.4.1.1/docs/Apecs-Concurrent.html. Accessed 2024-10-09.

Jonas Carpay. 2019. Apecs Shmup example. Source https://github.com/jonascarpay/apecs/blob/be72edab17eaaf9f771e45392b3837d31d8ff663/examples/Shmup.lhs#L338-L350. Accessed 2024-10-16.

Jonas Carpay. 2024a. Apecs.TH, makeWorld (v0.9.6). Source https://hackage.haskell.org/package/apecs-0.9.6/docs/Apecs-TH.html#v:makeWorld. Accessed 2024-10-15.

Jonas Carpay. 2024b. Apecs.Util, nextEntity (v0.9.6). Source https://hackage.haskell.org/package/apecs-0.9.6/docs/src/Apecs.Util.html#nextEntity. Accessed 2024-10-09.

Gwihen Etienne. 1984. Linear extensions of finite posets and a conjecture of G. Kreweras on permutations. *Discrete Math.* 52, 1 (mar 1984), 107–111. https://doi.org/10.1016/0012-365X(84)90108-0

Cédric Fournet and Georges Gonthier. 1996. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 372–385.

Cédric Fournet and Georges Gonthier. 2000. The join calculus: A language for distributed mobile programming. In *International Summer School on Applied Semantics*. Springer, 268–332.

Thomas Gillen. 2021. Legion: High performance entity component system (ECS) library. Releases https://crates.io/crates/legion. Living https://github.com/amethyst/legion. Accessed 2024-04-04.

Stephen Gutekanst. 2022. Let's build an Entity Component System (part 2): databases. Blog https://devlog.hexops.com/2022/lets-build-ecs-part-2-databases/. Accessed 2024-10-15.

Lars I. Hatledal, Yingguang Chu, Arne Styve, and Houxiang Zhang. 2021. Vico: An entity-component-system based co-simulation framework. *Simulation Modelling Practice and Theory* 108 (2021), 102243. https://doi.org/10.1016/j.simpat.2020.102243

Michael Hicks. 2015. What is PL research and how is it useful? http://www.pl-enthusiast.net/2015/05/27/what-is-pl-research-and-how-is-it-useful/

G. Kahn. 1974. The Semantics of a Simple Language for Parallel Programming. In *Information Processing '74: Proceedings of the IFIP Congress*, J. L. Rosenfeld (Ed.). North-Holland.

Eyal Kalderon. 2021. Amethyst Game Engine. Releases https://crates.io/crates/amethyst. Living https://github.com/amethyst/amethyst. Accessed 2024-10-09.

Indiana Kernick. 2020. EnTT Pacman. Source https://github.com/indianakernick/EnTT-Pacman/blob/8d0ad586decc80aebe1431456b092507be26b372/src/core/game.cpp/#L79-L89. Accessed 2024-10-16.

Lindsey Kuper and Ryan R. Newton. 2013. LVars: Lattice-based Data Structures for Deterministic Parallelism. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing* (Boston, Massachusetts, USA) *(FHPC '13)*. ACM, New York, NY, USA, 71–84. https://doi.org/10.1145/2502323.2502326

Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. 2014. Freeze After Writing: Quasi-deterministic Parallel Programming with LVars. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. ACM, New York, NY, USA, 257–270. https://doi.org/10.1145/2535838.2535842

Sandy Maguire. 2018. Ecstasy: A GHC.Generics based entity component system. Releases https://hackage.haskell.org/package/ecstasy. Accessed 2024-04-04.

Adam Martin. 2007a. Entity Systems are the future of MMOG development - Part 1. Blog https://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/. Accessed 2024-04-04.

Adam Martin. 2007b. Entity Systems are the future of MMOG development - Part 3. Blog https://t-machine.org/index.php/2007/12/22/entity-systems-are-the-future-of-mmog-development-part-3/. Accessed 2024-10-15.

Sander Mertens. 2020a. Building an ECS #2: Archetypes and Vectorization. Blog https://ajmmertens.medium.com/building-an-ecs-2-archetypes-and-vectorization-fe21690805f9. Accessed 2024-10-15.

Sander Mertens. 2020b. Making the most of ECS identifiers. Blog https://ajmmertens.medium.com/doing-a-lot-with-a-little-ecs-identifiers-25a72bd2647. Accessed 2024-10-09.

Sander Mertens. 2023. Why it is time to start thinking of games as databases. Blog https://ajmmertens.medium.com/why-it-is-time-to-start-thinking-of-games-as-databases-e7971da33ac3. Accessed 2024-10-15.

Sander Mertens. 2024a. Flecs: A fast entity component system (ECS) for C & C++. https://www.flecs.dev/flecs/. Living https://github.com/SanderMertens/flecs/. Accessed 2024-04-04.

Sander Mertens. 2024b. Flecs: Systems. https://www.flecs.dev/flecs/md_docs_2Systems.html#immediate-systems. Accessed 2024-10-16.

Mojang AB and Microsoft Corporation. 2024. Minecraft Attributions. https://www.minecraft.net/en-us/attribution. Accessed 2024-10-9.

Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. 2008. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *FSTTCS*.

Aleksandar Prokopec, Heather Miller, Tobias Schlatter, Philipp Haller, and Martin Odersky. 2012. FlowPools: a lock-free deterministic concurrent dataflow abstraction. In *LCPC*.

Thibault Raffaillac and Stéphane Huot. 2019. Polyphony: Programming Interfaces and Interactions with the Entity-Component-System Model. *Proc. ACM Hum.-Comput. Interact.* 3, EICS, Article 8 (jun 2019), 22 pages. https://doi.org/10.1145/3331150

Lucas Sardois. 2021. Generational indices guide. Blog https://lucassardois.medium.com/generational-indices-guide-8e3c5f7fd594. Accessed 2024-10-09.

Thomas Schaller. 2022. LazyUpdate in specs::world. Documentation https://docs.rs/specs/0.20.0/specs/world/struct.LazyUpdate.html. Accessed 2024-10-16.

Thomas Schaller. 2023. The Specs Book. https://amethyst.github.io/specs/docs/tutorials/. Living https://github.com/amethyst/specs/tree/master/docs/tutorials. Accessed 2024-04-04.

Lukas Schmierer. 2017. Benchmarks of various Rust Entity Component Systems. https://github.com/lschmierer/ecs_bench. Accessed 2024-10-9.

L. G. Tesler and H. J. Enea. 1968. A language design for concurrent processes. In *AFIPS (Spring)*.

Unity Technologies. 2024. ECS for Unity. https://unity.com/ecs. Version 0.17 https://docs.unity3d.com/Packages/com.unity.entities@0.17/manual/index.html. Version 1.2 https://docs.unity3d.com/Packages/com.unity.entities@1.2/manual/index.html. Accessed 2024-04-04.

Catherine West. 2018. Using Rust for Game Development (and What You Can Learn From It). Presentation https://kyren.github.io/rustconf_2018_slides/index.html. Video https://www.youtube.com/watch?v=aKLntZcp27M. Blog https://kyren.github.io/2018/09/14/rustconf-talk.html. Accessed 2024-04-04.

Mick West. 2007. Evolve Your Hierarchy. Blog https://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/. Accessed 2024-04-05.

A APPENDIX

The underlying language used in our running examples is a lambda calculus extended with conveniences. Its full grammar is as follows.

$n : \mathbb{Z}$	::=	$\ldots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \ldots \mid -n \mid n + n' \mid \lceil n/n' \rceil$	Integer
$e : E$	::=	$e_0 \mid e_1 \mid e_2 \mid \ldots$	Entity
b	::=	$n \stackrel{?}{=} n' \mid n \stackrel{?}{\neq} n' \mid b \wedge b'$	Bool
d	::=	... not otherwise mentioned ...	Binding
p	::=	$d \mid (d, (d', \ldots)) \mid (\langle d, \ldots \rangle, \langle (d', \ldots), \ldots \rangle)$	Pattern
$t : T$	::=	$d \mid t\ t' \mid \lambda p.\ t \mid m \mid n \mid e \mid$ **if** b **then** t **else** t'	Term