Apecs: A Type-Driven Entity-Component-System Framework

Jonas Carpay

***Abstract*—Despite the fact that object-oriented languages remain popular for game development, game programming has many unique challenges that are not easily solved by object-oriented design patterns. *Entity-Component-System* (ECS) frameworks are an interesting alternative to the object-oriented paradigm, remarkably well-suited for those challenges. Unfortunately, most programming languages struggle to support the ECS style of programming, and current implementations tend to suffer in terms of expressivity or performance.**

In this paper we present *apecs*, a type-driven ECS framework embedded in the Haskell programming language. It exposes a high-level domain-specific language that translates to fast primitive storage operations. The resulting code is concise, safe, and has excellent performance. This paper details the use of apecs, its implementation, its performance, and how to extend it.

I. Introduction

Game programming has many unique challenges. A commercial game engine consists of many subsystems like physics, input handling, rendering, and artificial intelligence, each operating under tight performance constraints. To complicate matters further, every object in the game world has a different way of interacting with both other game objects and the engine's subsystems. Game objects do not generally fall into a class hierarchy, and they are notoriously difficult to abstract. For example, is an invisible wall a wall that you cannot see, or is a normal wall an invisible wall that you do see? What about a fake wall, that you see but can move through, or a wall that moves? These issues make traditional design patterns susceptible to multiple-inheritance problems.

Modern commercial game engines solve this by using a composition-based approach instead, commonly called a *Component Pattern* [8]. This pattern defines a single general-purpose game object, the behaviour of which is altered by adding different components to it. Typical examples of component classes are ones that describe how an object is rendered, how its physics work, or how it is controlled. The wall from the example above would be assembled from e.g. a `WallModel` and a `WallBody`, and removing either would give us an invisible wall or fake wall, respectively.

Compared to traditional Object-Oriented design approaches, component patterns allow code reuse without risking multiple-inheritance problems. More importantly, for every game object, the programmer can easily make a near-infinite number of variations, by instantiating it with different components. This expressivity is important, as games programming is primarily a creative endeavor.

Despite their ubiquity, component patterns still have a number of significant problems. While it is now much easier to describe individual objects, writing game logic has arguably become more difficult. It is not obvious how game logic should be distributed over components, and communication between components can require complex traversals of the object graph. Furthermore, components themselves can prove just as difficult to abstract as the objects were initially.

TABLE I
Visualization of an example game state in an ECS framework

Entity	Position	Velocity	Player	Model
0	1.0	1.0	Player	CharacterModel
1	2.5	–	–	WallModel
2	9.0	-2.0	–	CharacterModel
3	–	–	–	–
...				

TABLE II
Game state from Table I after adding velocities to position

Entity	Position	Velocity	Player	Model
0	2.0	1.0	Player	CharacterModel
1	2.5	–	–	WallModel
2	7.0	-2.0	–	CharacterModel
3	–	–	–	–
...				

A. ECS Patterns

The *Entity-Component-System* (ECS) pattern is a game engine architecture inspired by the component pattern. In an ECS framework, the entire game state can be visualized as a table, as in Table I.

The rows represent *Entities*, and the colums represent *Components*. In the example above, Entity 1 has 2 Components; it has a position of 0, and is rendered as a `WallModel`. Entities are just integers, used to index the world state. Components are just pieces of data, and in contrast to the traditional component pattern, they do not typically have any code or associated methods. They can be freely added or removed from Entities at runtime.

In contrast to typical component patterns, an Entity in ECS is always said to "exist", it is never explicitly instantiated. Entity 3, or even Entity 10000, are both just as valid as Entity 2, they just do not have any components yet.

Game logic in an ECS framework is expressed entirely in terms of *Systems*. A System performs some operation globally on the game world. The typical example is adding an Entity's velocity to its position, for every Entity that has both of those Components, as in Table II.

ECS frameworks are a remarkably good fit for game programming. Like in component patterns, the programmer can easily encode common properties of game objects, without restricting how they are different from one another, just by adding the right Components. Systems provide a clear way to express the behaviour of these objects in general statements like “objects with a position and a velocity move by adding their position to their velocity”, solving the code organization issues of the component pattern. Finally, because Systems can operate directly on the relevant data, as we will see later, they are extremely fast and easy to parallelize.

B. Contributions

In this paper we present *apecs*, a simple yet powerful ECS framework embedded in Haskell. It leverages Haskell’s strong type system to build an elegant DSL from a small number of primitive storage operations. We expose the `cmap` combinator to concisely define Systems that get compiled into tight loops, while eliminating all boilerplate. The type system ensures that these Systems are well-formed and safe, unless explicitly requested otherwise. Interestingly, because Haskell makes no distinction between data and functions, we can even model the original component pattern by having Components hold functions.

It is important to stress that the apecs DSL is stateful and imperative (and arguably dynamically-typed), in contrast to most Haskell libraries. The goal is not to explore how to idiomatically write games in functional languages, but rather to present an elegant DSL suitable for high-performance applications.

C. Related Work

There is a large body of informal research on ECS in the form of blog posts and talks, and there are many open-source libraries available, each with its own interpretations of the paradigm. Arguably the first example was presented in 2002 [2], and describes a game engine in which the data is kept in a large, external database. Game logic would be expressed in a total of three separate programming languages. Modern libraries now generally store the Components in separate homogeneous data structures.

Our work was inspired by *specs* [11], an ECS framework written in the Rust programming language. Specs is fast and expressive, but like other ECS libraries, the Rust language does not easily facilitate the ECS paradigm. This has many implications, most notably that in all current ECS implementations, defining a System requires the user to also define error-prone boilerplate code for retrieving the right data.

While Rust is heavily inspired by Haskell, it has a more restrictive type system. Apecs uses features like type classes with multiple parameters, which Rust does not currently support. That said, work on types in Rust is still ongoing with work like Associated Types [9]. It might be possible to port apecs to Rust in the future, with all the associated benefits of using a more systems programming-oriented language.

Another programming paradigm used for games programming in Haskell is Functional Reactive Programming (FRP) [6]. FRP explores an idiomatic way to incorporate interactivity and time-dependence into functional programs. ECS, on the other hand, is a data-first imperative model, and seems to be entirely limited to games programming. The two solve entirely different problems, but one does not necessarily preclude the other.

D. Outline

We will start by giving a tour of the user-facing DSL exposed by apecs in the next section. The Section III will focus on how this DSL gets translated to smaller primitive operations, and how those operations are implemented. Section IV will investigate the performance of apecs, and finally Section V its shortcomings, and suggestions for future work.

Since people unfamiliar with Haskell have expressed interest in using apecs, we aim to keep Section II accessible to both people interested in functional programming, and people interested in game development. If you have no background in Haskell but want to write games using the apecs ECS framework, considering reading a Haskell introductory text like [7] or [1] up until monads and type classes before heading into Section III.

II. A Tour of Apecs

In this section, we will give a brief overview of the apecs DSL, focusing exclusively on how its high-level combinators are used. The details of their types and implementation are left until the next section. We will assume a simple game state like that of Table I. For more fleshed out examples of full games, refer to the project’s GitHub page [3].

A. The `cmap` Combinator

We start with the observation that the System “add every Entity’s Velocity to its Position” from the example, can be thought of in terms of a single function:

```
stepFun :: (Position, Velocity) -> Position
stepFun (Position p, Velocity v) = Position (v+p)
```

In fact, it turns out that this is true for most useful Systems. We base our DSL on this principle, for two reasons: First, in terms of expressivity, Systems are defined in the simplest possible terms; a pure function that they execute. Second, note that the function input’s type (here `(Position,Velocity)`) determines what Entities to iterate over and what Components to read, and the output type (here `Position`) determines what Components to write. That means that from just the function definition we have a lot of useful information at compile-time, which we can leverage for performance benefits.

In apecs, we turn a function into a System with `cmap` combinator, the primary construct of apecs:

```
stepSys :: System World ()
stepSys = cmap stepFun
```

This is generally written inline using a λ-abstraction:

```
cmap $ \(Position p, Velocity v) -> Position (v+p)
```

B. Component Composition

Both the input and output of the function used by `cmap` are Components. In its simplest form, this looks like e.g.

```
cmap $ \(Position p) -> Position (p+1)
```

where both the input and the output is the `Position` Component. As you can see in the previous example, however, a tuple of two Components like `(Position,Velocity)` is also considered a Component. The tuple represents the logical product, or intersection, of two (or more) Components.

Tuples can also occur in the output position to write multiple Components at once. This will reset the Positions and Velocities of every Entity with a Position, for example:

```
cmap $ \(Position _) -> (Position 0, Velocity 0)
```

By the same mechanisms that allow us to define tuple instances, we can define other useful compositions. The two primary examples are `Maybe c` which represents optionality, and `Not c` for negation. When used in the output type, `Maybe c`, for example, can be used to optionally delete a component.

```
-- Remove an Entity's PhysicsBody if out of bounds
cmap $ \(Position p, body)
       -> if inbounds p then Just body else Nothing
```

Similarly, a `Not c` in the input type will exclude that Component from a `cmap`:

```
  -- Move only non-player entities
  cmap $ \( Position p, Velocity v, Not :: Not
      Player)
         -> Position (p+v)
```

We will study these in more detail in Section III.

C. Unit Types

A common pattern is to use unit types (types that hold only a single value) to tag Entities. The `Player` Component we have been using so far is an example of this; it could be defined as

```
data player = Player
```

Tagging an entity is a matter of adding a `Player` to it. This is analogous to e.g. setting an object's `isPlayer = true`. Unit types are especially useful when used with `cmap`, like when directly targeting the player:

```
cmap $ \Player -> Velocity 0
```

D. Monadic composition

It might not surprise you to learn that in apecs, a System is a monad. This allows us to leverage Haskell's full support for monads. Most of your Systems, for example, will be written in do-notation, and lift side-effects into the IO monad:

```
runGame = do
  liftIO $ putStrLn "Hello␣World!"
  cmap $ \(Position p, Velocity v) -> Position (p+v)
```

We can monadically map over Entities using `cmapM` and `cmapM_`. In this example, we use nested `cmapM_`s to print the player's distance to other Entities:

```
cmapM_ $ \(Position pa, Player) -> do
  cmapM_ $ \(Position pb, Not :: Not Player) ->
    liftIO $ print (pb-pa)
```

E. Entity Creation

The primary method for creating a new Entities is through the `newEntity` function. We could replicate the game state of Table I like this, for example:

```
initGame = do
  newEntity
    (Position 1, Velocity 1, Player, CharacterModel)
  newEntity (Position 2.5, WallModel)
  newEntity
    (Position 9, Velocity (-2), CharacterModel)
```

Each successive call to `newEntity` will set the Components of a subsequent Entity, counting upwards from 0. The details of `newEntity` and how it stores its counting value in Section III-F

F. The `cfold` combinator

If `cmap` performs a general operation on the game world, `cfold` asks a question about the world. For example, we could count the number of entities that satisfy some condition as follows:

```
count p = cfold (\n c -> if p c then n+1 else n) 0
```

Like other `fold` functions, it takes an accumulation function of type `a -> c -> a` and an initial value of type `a`. It then iterates over every Component `c` , producing a final `a`.

G. Defining Composite Data

The compositional approach to Components is also useful for defining the data in a game. We can use type synonyms to assign names to common combinations of Components, and define reusable composite Components:

```
type Movable  = (Position, Velocity)
defaultMovable :: Movable
defaultMovable = (Position 0, Velocity 0)

type Kinetic = (Movable, Model)
defaultCharacter :: Kinetic
defaultCharacter = (defaultMovable, CharacterModel)

player :: (Kinetic, Player)
player = (defaultCharacter, Player)
```

These are just synonyms, and therefore completely free; we do not need to define any additional instances. For example, consider how they can be used to simplify `cmap` operations:

```
cmap $ \(_ :: Kinetic) -> defaultMovable
```

This would set the Position and Velocity of every Entity that also has a Model back to the default.

III. Implementation

In the previous section, we took a high-level tour of apecs. This should have given you an impression of what sort of statements its DSL supports. To understand its implementation, we will take a bottom-up approach. Paragraphs III-A and III-B dissect Listing 1, the core functionality of apecs. We then implement this DSL in paragraphs III-C through III-E. The remainder of this section investigates ways to extend apecs.

Listing 1. Apecs core

```
newtype Entity = Entity Int

newtype System w a = System (ReaderT w IO a)

type family Elem s

class (Elem (Storage c) ~ c) => Component c where
  type Storage c

class Component c => Has w c where
  getStore :: System w (Storage c)

class ExplInit s where
  explInit :: IO s

class ExplGet s where
  explGet    :: s -> Entity -> IO (Elem s)
  explExists :: s -> Entity -> IO Bool

class ExplSet s where
  explSet :: s -> Entity -> Elem s -> IO ()

class ExplDestroy s where
  explDestroy :: s -> Entity -> IO ()

class ExplMembers s where
  explMembers :: s -> IO (Vector Entity)
```

A. Component Stores

We turn an ordinary data type into a Component by giving it an instance of the `Component` type class. When doing so, you have to specify the type of its `Storage`, e.g.:

```
newtype Position = Position Int

instance Component Position where
  type Storage Position = Map Position
```

In this example, Position is stored in a `Map Position`. The `Map c` store is defined in `Apecs.Stores`, and is the most straightforward of the stores exposed by that module. We will therefore limit our discussion to the `Map` store for now, but more examples can be found in section III-F.

The `Elem` type-level function gives the type element of an element of a store, e.g. `Elem` (Map c) = c. In order for a component definition to be valid, `Elem` and `Storage` have to be inverses of one another, as seen in the constraint on the Component class in Listing 1.

A Component store is a homogeneous collection of Components. The `Expl` family of classes define operations supported by the store.

- `explExists` checks whether a store has a Component, for some Entity index.
- `explGet` reads a Component from the store, for some Entity index. This is an unsafe operation, reading a Component that does not exist can return undefined values. We therefore use `explExists` to first check whether the Component exists. This process is generally abstracted away through `cmap` or `Maybe c`.
- `explSet` writes a Component to the store at some Entity index.
- `explDestroy` deletes a Component from the store.
- `explMembers` produces a list of all Entities for which the store has Components. An Entity is on the list if and only if it `explExists`.
- `explInit` produces a reference to a new, empty store.

This is the entire interface for a store, and as we will see later, some stores support only a subset of these functions. As you can see from the type signatures in Listing 1, these functions operate in the IO monad. The stores are regarded as mutable, the `s` argument is just a reference to the mutable store. There is a discussion on the choice for the IO monad in Section V-A.

B. Systems

The prefix `expl` stands for explicit, indicating that each of these functions takes an explicit store argument. The primary use of the `System` monad is to erase this argument in the DSL, instead inferring it from type signatures. Each of the primitive operations (except `explInit`) has a counterpart in the `System` monad. For example, `ExplGet` and its member functions have these:

```
type Get w c = (Has w c, ExplGet (Storage c))

exists :: forall w c. Get w c
       => Entity -> Proxy c -> System w Bool
exists ety _ = do
  s :: Storage c <- getStore
  liftIO (explExists s ety)

get :: forall w c. Get w c => Entity -> System w c
get ety = do
  s :: Storage c <- getStore
  liftIO (explGet s ety)
```

We see the inference of the store in action in the use of `getStore`. It is the sole member function of the `Has w c` type class, which produces a store for Component `c` from some context `w` (for world). Most Systems exposed by apecs follow this pattern of first producing the correct store with `getStore`, and then lifting some operation on the store into the IO monad.

We can now use `get` in Systems like this:

```
runGame :: Get w Position
        => System w ()
runGame = do
  Position p <- get 0
  liftIO $ print p
```

In this case, `get` knows what store to retrieve from its return type of `Position`. In the cases of `exists`, however, we indicate what Component we are talking about by using a `Proxy :: Proxy c` to pass a phantom `c` argument.

The `Get` type synonym combines the constraints `Has` and `ExplGet` into one, as they typically occur together. We similarly have `Set`, `Members`, and `Destroy`.

While we now have direct control over our data stores, working with Entity values is still error-prone. That's why combinators like `cmap` allow us to hide this from us as much as possible. Unfortunately, `cmap` cannot be used to express all game logic, so we occasionally resort to these intermediate functions.

C. High-level Combinators

Now that we know what primitive functions exist and how they relate to the `System` monad, we are ready to define `cmap`. Let's start by looking at its type signature:

```
cmap :: forall w cx cy.
        (Get w cx, Members w cx, Set w cy)
     => (cx -> cy) -> System w ()
```

For a function to be `cmap`ped, we need `Get` and `Members` instances for its `cx` Component, and a `set` instance for `cy`. This already hints at its implementation:

```
cmap f = do
  sx :: Storage cx <- getStore
  sy :: Storage cy <- getStore
  liftIO$ do
    sl <- explMembers sx
    forM_ sl $ \ e -> do
      x <- explGet sx e
      explSet sy e (f x)
```

Note that we could equivalently define `cmap` entirely in System:

```
cmap f = do
  sl <- members (Proxy :: Proxy cx)
  forM_ sl $ e -> do
    x <- get e
    set e (f x)
```

This definition shows more clearly what is happening; we first produce a list of members for `cx`. For each member, we then read its `cx` value, and write back the result `f x :: cy`. Since this second definition makes two calls to `getStore` ever iteration (once in `get`, and once in `set`), we use the first definition instead.

D. The Game World

Recall the `Has w c` type class, which produces a store for Component `c` for some world `w`. So far, we have not yet touched on the definition of the game world, the `w` argument. This is generally done by first declaring the games' components:

```
newtype Position = Position Double
newtype Velocity = Velocity Double

instance Component Position where
  type Storage Position = Map Position
instance Component Velocity where
  type Storage Velocity = Map Velocity
```

We then define a world that contains the stores for these Components. For each Component, we also define an instance of `Has`, in order to access the Component from a System. Finally, we need a function to initialize our game world when starting the game, using the individual stores' `ExplInit`:

```
data World = World
  { positions  :: Storage Position
  , velocities :: Storage Velocity
  }

instance Has World Position where
  getStore = System (asks positions)
instance Has World Velocity where
  getStore = System (asks velocities)

initWorld = do
  sp <- explInit
  sv <- explInit
  return $ World sp sv
```

It is important to understand how a game world is defined, but manually writing the definitions is tedious and error-prone. We therefore generally automate this using Template Haskell.

```
newtype Position = Position Double
newtype Velocity = Velocity Double

instance Component Position where
  type Storage Position = Map Position
instance Component Velocity where
  type Storage Velocity = Map Velocity

makeWorld "World" [''Position, ''Velocity]
```

E. Composition

Now that we know how the type-level machinery works, we can look at interesting compositions of Components.

1) Tuples: As mentioned earlier, tuples of Components are also Components. Because every Component has a unique `Storage`, we necessarily also have a tuple of stores, and ways to produce those from the game world:

```
instance (Component a, Component b)
      => Component (a, b) where
  type Storage (a,b) = (Storage a, Storage b)

instance (Has w a, Has w b) => Has w (a,b) where
  getStore = liftM2 (,) getStore getStore

type instance Elem (a,b) = (Elem a, Elem b)
```

Calling `getStore` on a tuple `(ca,cb)` gives a tuple of their respective stores. A store of tuples supports all the primitive storage operations, with the exception of `ExplInit`:

```
instance (ExplGet a, ExplGet b) => ExplGet (a, b)
instance (ExplSet a, ExplSet b) => ExplSet (a, b)
instance (ExplDestroy a, ExplDestroy b)
          => ExplDestroy (a, b)
instance (ExplMembers a, ExplGet b)
          => ExplMembers (a, b)
```

We previously touched upon setting and getting a Component, and destroying one is similarly straightforward. The other two functions, `(expl)Exists` and `(expl)Members` deserve some special attention. For `explExists`, the tuple returns whether the Component is present in both of its constituent stores.

The type signature of the `ExplMembers` instance hints at its implementation; it produces a list of members of Component *a* `explMembers` , and uses `explExists` to filter those that also have a *b*. This results in a list of members that are guaranteed to have both Components.

This can have significant performance implications. Imagine we have 1000 Entities with a Position, but only one of which also has a Velocity. If we ask for the members of `(Position, Velocity)`, we first iterate over all Entities with Positions and filter out the one which also has a Velocity. Asking for the members of `(Velocity, Position)` , however, will only iterate over the single Velocity component, making it much faster. This means that in tuples, the rarest Component should generally occur first.

2) The `Not` Component: Defining `Not` is similar to defining tuples, but it has a dedicated store:

```
data Not c = Not
newtype NotStore s = NotStore s

instance Component c => Component (Not c) where
  type Storage (Not c) = NotStore (Storage c)

instance Has w c => Has w (Not c) where
  getStore = NotStore <$> getStore

type instance Elem (NotStore s) = Not (Elem s)
```

So when we request the store of a `Not c`, it will return the store for Component `c` wrapped in a `NotStore`. `NotStore` wraps around another store, upon which `explExists` will invert its result, `explGet` will always return `Not`, and `explSet` will actually destroy the underlying Component.

```
instance ExplGet s => ExplGet (NotStore s)
instance ExplDestroy s => ExplSet (NotStore s)
```

Note that we cannot iterate directly over a `Not`, because its store does not have a `ExplMembers` instance. The only lawful implementation would require us to iterate over all possible integer values, and is therefore not given. This has the additional effect that we can `cmap` over `(Position, Not Player)`, but not over `(Not Player, Position)`, since the first member of the tuple needs to support `members` in order for the entire tuple to do so.

3) The `Maybe` Component: The instances for `Maybe` are similar to those for `Not`:

```
newtype MaybeStore s = MaybeStore s

instance Component c => Component (Maybe c) where
  type Storage (Maybe c) = MaybeStore (Storage c)

instance Has w c => Has w (Maybe c)

type instance Elem (MaybeStore s) = Maybe (Elem s)
instance ExplGet s => ExplGet (MaybeStore s)
instance (ExplSet s, ExplDestroy s)
       => ExplSet (MaybeStore s)
```

Getting a `Maybe c` yields `Just c` if *c* exists, and `Nothing` otherwise, and `explExists` always returns True. While `Not` is primarily useful for filtering `cmap`, `Maybe` has interesting use cases by itself. For example, it can turn `get` into a safe operation:

```
runGame = do
  mPos <- get 0
  case mpos of
  | Just p -> liftIO $ print p
  | Nothing -> liftIO $
    putStrLn "Entity␣0␣does␣not␣have␣a␣position"
```

4) The `Entity` Component: The `Entity` type itself can be used as a Component, but it is not like other Components. It has a `Has` instance for every world, its store `EntityStore` holds no values, and only has a `ExplGet` instance. When querying the `EntityStore` for an Entity, it will simply return that Entity.

The primary use is in `cmap`, where we use it to access the current Entity index. For example, we can print the player's Entity like this:

```
cmapM_ $ \(Player, Entity e) ->
  liftIO $ print e
```

F. Other Stores

Our discussion up until now has only concerned the `Map` store. A `Map` has instances for all `Expl`-classes, and has the semantics generally expected of component stores. There are a number of other possible useful stores, that provide either different semantics or performance characteristics.

1) The `Global` store and the `EntityCounter`: For example `Global c` will only hold a single value, will always return True for `explExists`, and will ignore the Entity arguments of `explGet` and `explSet`.

It is used to define a single, global `EntityCounter`, used by `newEntity` to use a new Entity every time:

```
newEntity :: ( Set w c
             , Get w EntityCounter
             , Set w EntityCounter)
          => c -> System w Entity
newEntity c = do
  EntityCounter ety <- get 0
  set ety c
  set 0 $ EntityCounter (n+1)
  return ety
```

In this example, the `0` argument to `get` and `set` is ignored; a `Global` always operates on the single value it holds. It does not have instances for `ExplMembers` or `ExplDestroy`.

In order to facilitate `newEntity`, a world created through `makeWorld` will have an `EntityStore` in addition to its other stores.

2) The `Unique` Store: The `Unique` is a Component of which no more than one can ever exist. An example use case is the `Player` component from the example. If we are sure that there will only ever be one player, we can enforce this by storing the `Player` in a `Unique Player`.

3) The `Cache`: `Cache n s` wraps around another store `s` and adds a cache of size `n :: Nat`. By adding a sufficiently large cache to an ordinary map we can store all components in a fixed-size vector. This gives us $O(1)$ runtime for all primitive storage operations. If we wanted to store all Positions in a `Cache`, we could write

```
instance Component Position where
  type Storage Position = Cache 100 (Map Position)
```

The `Cache` shows how composition of stores can be used for great performance benefits. In fact, the effective caching of Components is what allows apecs to be fast.

4) Extending Stores: Because stores live in the IO monad, we can execute arbitrary side-effects when querying stores. This can be used to add almost any behaviour to a store, and is useful for defining your own specialized stores. For example, we could define a store for Positions that automatically keeps a mutable spatial hash of positions, allowing for efficient spatial queries. In fact, this is so common that apecs includes a small number of helper functions for spatial hashing.

A second possibility is to have stores that live in foreign data structures, like different game engines. For example, the apecs-physics library [4] internally uses the Chipmunk physics engine [5] written in C.

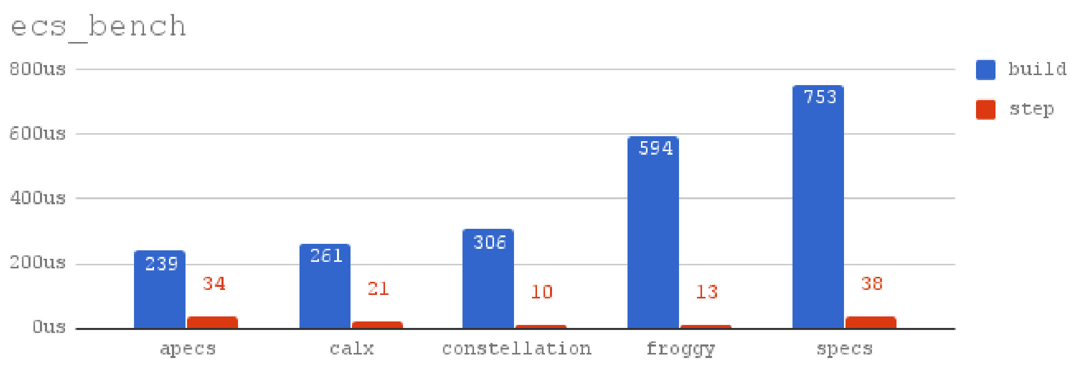

Fig. 1. Benchmark results of apecs and the libraries in `ecs_bench`. The vertical axis shows the running time, lower is better.

IV. Performance

It is difficult to compare the performance of popular ECS libraries, as they span a large range of languages and designs. The most prominent example of an ECS benchmark is `ecs_bench`, which compares the performance of libraries written in Rust [10]. It consists of two separate benchmarks. In the first, `build`, it creates 1000 entities with a Position and a Velocity, and another 9000 with just a Position. Both the Position and Velocity are two-dimensional vectors of doubles. The second benchmark, `step`, times how long it takes to add the Velocities to the Positions.

We base our own benchmarks on this same model, which should give a general estimate of relative performance. We take a number of precautions for improving benchmark results:

- We add `Caches` around the stores for Position and Velocity, of size 10000 and 1000, respectively.
- We write the mapped function as `stepFun (Velocity v, Position p) = Position (v+p)`, making sure to put Velocity in the *first* position. This way, `cmap` iterates over Velocity Components, of which there are fewer than Positions.
- We enable all GHC optimizations and compile using the LLVM backend.

The results of the benchmark can be found in Figure 1.

The benchmark for apecs was run using the `criterion` benchmarking library, the others are the results from running `ecs_bench`. We should be careful to draw any strong conclusions from this data, for a number of reasons. For example, the benchmark is not necessarily reflective of real-world performance, it says nothing about memory usage, and `ecs_bench` has not been updated in a long time. Still, it seems like the performance of apecs is at least competitive with that of similar Rust libraries. Since Rust is generally more suited to low-level programming than Haskell, this is a promising result.

V. Conclusion

We first introduced a domain-specific language for expressing game logic, and then showed how to construct that DSL from primitive storage operations. Apecs shows a promising implementation of the ECS paradigm, ready to be used for game development in Haskell. By choosing the right stores we can even compete with more low-level languages in terms of performance, while writing significantly less code.

A. Future Work

There are a number of avenues for further improvements.

1) Performance Tuning: While apecs offers good performance, it has not been extensively tuned. GHC is capable of producing extraordinarily fast code under the right circumstances, so it is possible that significant gains remain to be had. For example, the use of Compact Regions might allow more efficient garbage collection, or better use of strictness or reformulation of certain functions might lead to more GHC optimization.

2) System as a Monad Transformer: Apecs currently only runs in the IO monad. Initially, however, System was defined as a monad transformer, so the underlying monad was kept variable. This was abandoned because the resulting code was significantly slower than when the monad was specialized to IO.

Keeping the monad variable has allows for a number of interesting options. For example, we could run Systems atomically by running them in the STM monad for easy, safe parallelism. Some early work suggests that these performance issues are no longer present in newer GHC versions.

References

[1] C. Allen and J. Moronuki. *Haskell Programming from First Principles*. Allen and Moronuki Publishing, 2016. ISBN: 9781945388033.

[2] Scott Bilas. “A Data-Driven Game Object System”. http://gamedevs.org/uploads/data-driven-game-object-system.pdf. Game Developers Conference.

[3] Jonas Carpay. *apecs*. https://github.com/jonascarpay/apecs. 2017.

[4] Jonas Carpay. *apecs-physics*. https://github.com/jonascarpay/apecs-physics. 2017.

[5] *Chipmunk2D*. https://github.com/slembcke/Chipmunk2D. 2018.

[6] Conal Elliott and Paul Hudak. “Functional Reactive Animation”. In: *International Conference on Functional Programming*. 1997. URL: http://conal.net/papers/icfp97/.

[7] Graham Hutton. *Programming in Haskell*. 2nd. New York, NY, USA: Cambridge University Press, 2016. ISBN: 1316626229, 9781316626221.

[8] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 2014. ISBN: 0990582906.

[9] Rust. *Generic Associated Types*. RFC 1598. Apr. 2016. URL: https://github.com/rust-lang/rfcs/blob/master/text/1598-generic_associated_types.md.

[10] Lukas Schmierer. *ecs_bench*. https://github.com/lschmierer/ecs_bench. 2017.

[11] *Specs Parallel ECS*. https://github.com/slide-rs/specs. 2018.