Exploring Codata: The Relation to Object-Orientation

ZACHARY SULLIVAN, University of Oregon

Functional languages are known to enjoy an elegant connection to logic: lambda-calculus corresponds to natural deduction. Unfortunately, the same cannot be said for object-oriented languages. Type systems have been designed to capture all the fancy features present in current object-oriented languages. We believe, however, that the logical foundation of object-orientation has not yet been fully explored. Our goal is to describe how objects arise naturally in logic.

The dual of data, known as codata, can be seen as an object. Whereas data is constructed, codata specifies all the possible ways to use it. Analogously to an object defining a pair, a codata pair gives two methods for accessing the first and second components.

To substantiate this approach we extend a basic language containing both data and codata with the essential object-oriented features of subtyping, classes, and inheritance. This suggests how the functional and object-oriented paradigms can embrace each other; the extended language also provides a suitable intermediate language for the compilation of the two programming paradigms.

Additional Key Words and Phrases: Codata, Curry-Howard, Object-orientation

1 INTRODUCTION

The Curry-Howard correspondence is not only the coincidence that programming language researchers and logicians develop similar systems independently, but it can be used as a tool for information to be shared between the two communities. For instance, linear type systems [24] were inspired by linear logic [11]. Linear logic's rules prevent multiple uses of an assumption, which when incorporated into a type system, can solve the problem of unwanted duplication of resources in a program. One such area of programming languages that has not seen much sharing of ideas with logic is object-oriented programming. Despite its popularity as a methodology for abstraction and structuring of code, it is unclear what logic lies at its foundation. That is not to say that strong type systems have not been developed to verify the safety of these languages [1, 4, 14]. These solutions start with an object-oriented language and work back to logic, but what can we learn from travelling in the other direction?

Like with linear type systems, we start our search for a logical foundation with linear logic. Linear logic has been shown to have a Curry-Howard connection to polarized programming languages [18, 26] and call-by-push-value [17]. An important aspect of these languages is that they contain two notions of product: one that focuses on the introduction and another that focuses on elimination. The former maps cleanly to *data* types found in functional languages, where the programmer defines a set of constructors that introduce an instance of the type. The latter maps to *codata* types, where the programmer defines a set of observations, or projections, that eliminate the type. Of course, defining a type by its observations captures two basic aspects of *objects* from object-oriented programming, where *fields* are data that can be projected from an object and *methods* are requests that an object computes.

Though the notion of codata is similar to objects, the gap between these two language features still appears large. This is in part because object-oriented languages contain extra ideas not found in codata, such as subtyping, classes, and inheritance. Additionally, objects have been around longer and discussed much more than the relatively mysterious codata. Because of these facts, it comes as no surprise that the connection between the two has not received attention. In this paper, we aim to explore the relationship of codata and objects as follows:

- In order to give an intuition for what codata is, Section 2 introduces codata through examples. Throughout, we make comparisons with objects to clearly convey the similarities of the two language features.
- In a first step towards a Curry-Howard correspondence between logic and object-orientation, Section 3 gives a description of $\mathcal{H}$: a language that combines data and codata.
- Since subtyping is one of the defining features of object-oriented languages, Section 4 extends $\mathcal{H}$ to include subtyping for codata leading to $\mathcal{H}_{<:}$.
- Section 5 recalls Featherweight Java [14], a standard model of an object-oriented language. The language contains the popular features of subtyping, classes, inheritance, and type casting. It has been used to formalize extensions to languages like Java and C#.
- Section 6 presents a compilation of Featherweight Java into $\mathcal{H}_{<:}$. We find that *classes* can be compiled into codata types (which specify the class's interface) and constructor functions (which encode the object's constructor).
- Finally, Section 7 describes how thinking of objects as codata can help us extend object-oriented languages with algebraic data types, fully corecursive objects, and the ability to introduce objects with copattern matching. Additionally, we describe how $\mathcal{H}_{<:}$ can be seen as an intermediate language for both a functional and object-oriented language. So bridging the gap between codata and objects also provides a common ground between these two popular programming paradigms.

2 USING CODATA

This section contains co-authored material [9]. I am responsible for the database and game examples that appear here in the paper. Additionally, I am responsible for all of the online code containing codata examples.

We start by presenting codata as a merger between theory and practice, whereby *encoding* data types by their observations (*i.e.* with codata) turns out to be a useful intermediate step in the usual encodings of data in the λ-calculus. *Demand-driven programming* is considered a virtue of lazy languages, but codata is an evaluation-strategy-independent tool for capturing this programming idiom. Codata exactly captures the essence of *procedural abstraction*, as achieved with λ-abstractions and objects, with a logically founded formalism [10]. Strengthening the type system to include indexed and self-referential types allows codata to express *pre- and post- conditions* found in protocols and Hoare-style logic [23]; this feature is available in some object systems via type-state guarded method calls [8].

2.1 Encoding Data with Codata

Crucial information structures, like booleans, numbers, lists, and grammars can be encoded in the untyped λ-calculus (*a.k.a.* Church encodings) or in the typed polymorphic λ-calculus (*a.k.a.* Böhm-Berarducci [6] encodings). It is quite remarkable that data structures can be simulated with just first-class, higher-order functions. The downside is that these encodings can be obtuse at first blush, and have the effect of obscuring the original program when *everything* is written with just λs and application. For example, the λ-representation of the boolean value true, the first projection out of a pair, and the constant function K are all expressed as $\lambda x.\lambda y.x$, which is not that immediately evocative of its multi-purpose nature.

Object-oriented programmers have also been representing data structures in terms of objects. This is especially visible in the Smalltalk lineage of languages like Scala, wherein an objective is that everything that can be an object is. As it turns out, the object-oriented features needed to perform this representation technique are *exactly* those of codata. That is because Church-style encodings and object-oriented representations of data all involve *switching focus from the way values are built (i.e. introduced) to the way they are used (i.e. eliminated).*

Consider the representation of Boolean values as an algebraic data type. There may be many ways to use a `Boolean` value. However, it turns out that there is a *most-general* eliminator of `Boolean`s: the expression `if b then x else y`. This basic construct can be used to define all the other uses for `Bool`s. Instead of focusing on the constructors `True` and `False` let's focus on this most-general form of `Bool` elimination; this is the essence of the encodings of booleans in terms of objects. In other words, booleans can be thought of as objects that implement a single method: `If`. So that the expression `if b then x else y` would instead be written as

```
b.If(x,y)
```

We then define the true and false values in terms of their reaction to `If`:

```
true = {If(x,y) → x}                    false = {If(x,y) → y}
```

Or alternatively, we can write the same definition using copatterns, popularized for use in the functional paradigm by Abel *et.al.* [2] by generalizing the usual pattern-based definition of functions by multiple clauses, as:

```
true.If(x,y) = x                        false.If(x,y) = y
```

This works just like equational definitions by pattern-matching in functional languages: the expression to the left of the equals sign is the same as the expression to the right (for any binding of `x` and `y`). Either way, the net result is that

```
true.If("yes","no") = "yes"       false.If("yes","no") ="no"
```

This covers the object-based presentation of booleans in a dynamically typed language, but how do static types come into play? In order to give a type description of the above boolean objects, we can use the following interface, analogous to a Java interface:

```
codata Bool where If : Bool → (forall a. a → a → a)
```

This declaration is dual to a data declaration in a functional language: data declarations define the types of constructors (which produce values of the data type) and codata declarations define the types of destructors (which consume values of the codata type) like `If`. In this case, the reason that the `If` observation introduces its own polymorphic type `a` is because an if-then-else might return any type of result (as long as both branches agree on the type). That way, both the two objects `true` and `false` above are values of the codata type `Bool`.

At this point, the representation of booleans as codata looks remarkably close to the encodings of booleans in the λ-calculus. Indeed, the only difference is that in the λ-calculus we "anonymize" booleans. Since they reply to only one request, that request name can be dropped. We then arrive at the familiar encodings in the polymorphic λ-calculus:

$$Bool = \forall a.a \to a \to a \qquad true = \Lambda a.\lambda x{:}a.\lambda y{:}a.x \qquad false = \Lambda a.\lambda x{:}a.\lambda y{:}a.y$$

In addition, the invocation of the `If` method just becomes ordinary function application; `b.If(x,y)` of type a is written as $b\ a\ x\ y$. Otherwise, the definition and behavior of booleans as either codata types or as polymorphic functions are the same.

2.2 Demand-Driven Programming

In "Why functional programming matters" [12], Hughes motivates the utility of practical functional programming through its excellence in compositionality. When designing programs, one of the goals is to decompose a large problem into several manageable sub-problems, solve each sub-problem in isolation, and then compose the individual parts together into a complete solution. Unfortunately, Hughes identifies some examples of programs which resist this kind of approach.

In particular, numeric algorithms—for computing square roots, derivatives integrals—rely on an infinite sequence of approximations which converge on the true answer only in the limit of the sequence. For these numeric algorithms, the decision on when a particular approximation in the sequence is "close enough" to the real answer lies solely in the eyes of the beholder: only the observer of the answer can say when to stop improving the approximation. As such, standard imperative implementations of these numeric algorithms are expressed as a single, complex loop, which interleaves both the concerns of producing better approximations with the termination decision on when to stop. Even more complex is the branching structure of the classic minimax algorithm from artificial intelligence for searching for reasonable moves in two-player games like chess, which can have an unreasonably large (if not infinite) search space. Here, too, there is difficulty separating generation from selection, and worse there is the intermediate step of pruning out uninteresting sub-trees of the search space (known as alpha-beta pruning). As a result, a standard imperative implementation of minimax is a single, recursive function that combines all the tasks—generation, pruning, estimation, and selection—at once.

Hughes shows how both instances of failed decomposition can be addressed in functional languages through the technique of *demand-driven programming*. In each case, the main obstacle is that the control of how to drive the next step of the algorithm—whether to continue or not—lies with the consumer. The producer of potential approximations and game states, in contrast, should only take over when demanded by the consumer. By giving primary control to the consumer, each of these problems can be decomposed into sensible sub-tasks, and recomposed back together. Hughes uses lazy evaluation, as found in languages like Miranda and Haskell, in order to implement the demand-driven algorithms. However, the downside of relying on lazy evaluation is that it is a whole-language decision: a language is either lazy by default, like Haskell, or not, like OCaml. When working in a strict language, expressing these demand-driven algorithms with manual laziness loses much of their original elegance [20].

In contrast, a language should directly support the capability of yielding control to the consumer independently of the language being strict or lazy; analogously to what happens with lambda abstractions. An abstraction computes on-demand, why is this property relegated to this predefined type only? In fact, the concept of *codata* also has this property. As such, it allows us to describe demand-driven programs in a way that works just as well in Haskell as in OCaml without any additional modification. For example, we can implement Hughes' demand-driven AI game in terms of codata instead of laziness. To represent the current game state, and all of its potential developments, we can use an arbitrarily-branching tree codata type.

```
codata Tree a where
  Node     : Tree a → a
  Children : Tree a → List (Tree a)
```

The task of generating all potential future boards from the current board state produces one of these tree objects, described as follows (where `moves` of type `Board → List Board` generates a list of possible moves):

```
gameTree : Board → Tree Board
(gameTree b).Node     = b
(gameTree b).Children = map gameTree (moves b)
```

Notice that the tree might be finite, such as in the game of Tic-Tac-Toe. However, it would still be inappropriate to waste resources fully generating all moves before determining which are even worth considering. Fortunately, the fact that the responses of a codata object are only computed when demanded means that the consumer is in full control over how much of the tree is generated, just as in Hughes' algorithm. This fact lets us write the following simplistic prune function which cuts off sub-trees at a fixed depth.

```
prune : Int → Tree Board → Tree Board
(prune x t).Node     = t.Node
(prune 0 t).Children = []
(prune x t).Children = map (prune(x-1)) t.Children
```

The more complex alpha-beta pruning algorithm can be written as its own pass, similar to prune above. Just like Hughes' original presentation, the evaluation of the best move for the opponent is the composition of a few smaller functions:

```
eval = maximize . maptree score . prune 5 . gameTree
```

What is the difference between this codata version of minimax and the one presented by Hughes that makes use of laziness? They both compute on-demand which makes the game efficient. However, demand-driven code written with codata can be easily ported between strict and lazy languages with only syntactic changes. In other words, codata is a general, portable, programming feature which is the key for compositionality in program design.[1]

2.3 Abstraction Mechanism

In the pursuit of scalable and maintainable program design, the typical followup to composability is abstraction. The basic purpose of abstraction is to hide certain implementation details so that different parts of the code base need not be concerned with them. For example, a large program will usually be organized into several different parts or "modules," some of which may hold general-purpose "library" code and others may be application-specific "clients" of those libraries. Successful abstractions will leverage tools of the programming language in question so that there is a clear interface between libraries and their clients, codifying which details are exposed to the client and which are kept hidden inside the library. A common such detail to hide is the concrete representation of some data type, like strings and collections. Clear abstraction barriers give freedom to both the library implementer (to change hidden details without disrupting any clients) as well as the client (to ignore details not exposed by the interface).

Reynolds [21] identified, and Cook [7] later elaborated on, two different mechanisms to achieve this abstraction: abstract data types and procedural abstraction. Abstract data types are crisply expressed by the Standard ML module system, based on existential types, which serves as a concrete practical touchstone for the notion. Procedural abstraction is pervasively used in object-oriented languages. However, due to the inherent differences among the many languages and the way they express procedural abstraction, it may not be completely clear of what the "essence" is, the way existential types are the essence of modules. The language-agnostic representation of procedural abstraction *is codata*. The combination of observation-based interfaces, message-passing, and dynamic dispatch are exactly the tools needed for procedural abstraction. Other common object-oriented features—like inheritance, subtyping, encapsulation, and mutable state—are orthogonal to

[1]To see the full code for all the examples of [13] implemented in terms of codata, visit https://github.com/zachsully/codata_examples.

this particular abstraction goal. While they may be useful extensions to codata for accomplishing programming tasks, only pure codata itself is needed to represent abstraction.

Specifying a codata type is giving an interface—between an implementation and a client—so that instances of the type (implementations) can respond to requests (clients). In fact, method calls are the only way to interact with our objects. As usual, there is no way to "open up" a higher-order function—one example of a codata type—and inspect the way it was implemented. The same intuition applies to all other codata types. For example, Cook's [7] procedural "set" interface can be expressed as a codata type with the following observations:

```
codata Set where
  IsEmpty  : Set → Bool
  Contains : Set → Int → Bool
  Insert   : Set → Int → Set
  Union    : Set → Set → Set
```

Every single object of type `Set` will respond to these observations, which is the only way to interact with it. This abstraction barrier gives us the freedom of defining several different instances of `Set` objects that can all be freely composed with one another. One such instance of `Set` uses a list to keep track of a hidden state of the contained elements (where `elemOf : List Int → Int → Bool` checks if a particular number is an element of the given list, and the standard functional fold is `fold : (a → b → b) → b → List a → b`):

```
finiteSet : List Int → Set
(finiteSet xs).IsEmpty    = xs == []
(finiteSet xs).Contains y = elemOf xs y
(finiteSet xs).Insert   y = finiteSet (y:xs)
(finiteSet xs).Union    s = fold (λx t → t.Insert x) s xs

emptySet = finiteSet []
```

But of course, many other instances of `Set` can also be given. For example, this codata type interface also makes it possible to represent infinite sets like the set `evens` of all even numbers which is defined in terms of the more general `evensUnion` that unions all even numbers with some other set (where the function `isEven : Int → Int` checks if a number is even):

```
evens = evensUnion emptySet

evensUnion : Set → Set
(evensUnion s).IsEmpty    = False
(evensUnion s).Contains y = isEven y || s.Contains y
(evensUnion s).Insert   y = evensUnion (s.Insert y)
(evensUnion s).Union    t = evensUnion (s.Union t)
```

Because of the natural abstraction mechanism provided by codata, different `Set` implementations can interact with each other. For example, we can union a finite set and `evens` together because both definitions of `Union` know nothing of the internal structure of the other `Set`. Therefore, all we can do is apply the observations provided by the `Set` codata type.

While sets of numbers are fairly simplistic, there are many more practical real-world instances of the procedural abstraction provided by codata to be found in object-oriented languages. For example, databases are a good use of abstraction, where basic database queries can be represented as the observations on table objects. A simplified interface to a database table (containing rows of type a) with selection, deletion, and insertion, is given as follows:

```
codata Database a where
  Select : Database a → (a → Bool) → List a
  Delete : Database a → (a → Bool) → Database a
  Insert : Database a → a → Database a
```

On one hand, specific implementations can be given for connecting to and communicating with a variety of different databases—like Postgres, MySQL, or just a simple file system—which are hidden behind this interface. On the other hand, clients can write generic operations independently of any specific database, such as copying rows from one table to another or inserting a row into a list of compatible tables:

```
copy : Database a → Database a → Database a
copy from to = let rows = from.Select(λ_ → True)
               in foldr (λrow db → db.Insert row) to rows

insertAll : List (Database a) → a → List (Database a)
insertAll dbs row = map (λdb → db.Insert row) dbs
```

In addition to abstracting away the details of specific databases, both copy and insertAll can communicate between completely different databases by just passing in the appropriate object instances, which all have the same generic type. Another use of this generality is for testing. Besides the normal instances of Database a which perform permanent operations on actual tables, one can also implement a fictitious *simulation* which records changes only in temporary memory. That way, client code can be seamlessly tested by running and checking the results of simulated database operations that have no external side effects by just passing pure codata objects.

2.4 Representing Pre- and Post-Conditions

The extension of data types with indexes (*a.k.a.* generalized algebraic data types) has proven useful to statically verify a data structure's invariant, like for red-black trees [25]. With indexed data types, the programmer can inform the static type system that a particular value of a data type satisfies some additional conditions by constraining the way in which it was constructed. Unsurprisingly, indexed codata types are dual and allow the creator of an object to constrain the way it is going to be used, thereby adding Hoare-style pre- and post-conditions to the observations of the object. In other words, in a language with type indexes, codata enables the programmer to express more information in its interface.

This additional expressiveness simplifies applications that rely on a type index to guard observations. Thibodeau *et.al.* [23] give examples of such programs, including an automaton specification where its transitions correspond to an observation that changes a pre- and post-condition in its index, and a fair resource scheduler where the observation of several resources is controlled by an index tracking the number of times they have been accessed. For concreteness, let's use an indexed codata type to specify safe protocols as in the following example from an object-oriented language with guarded methods:

```
index Raw, Bound, Live

codata Socket i where
  Bind    : Socket Raw   → String → Socket Bound
  Connect : Socket Bound → Socket Live
  Send    : Socket Live  → String → ()
  Receive : Socket Live  → String
  Close   : Socket Live  → ()
```

This example comes from DeLine and Fähndrich [8], where they present an extension to C♯ constraining the pre- and post-conditions for method calls. If we have an instance of this `Socket i` interface, then observing it through the above methods can return new socket objects with a different index. The index thereby governs the order in which clients are allowed to apply these methods. A socket will start with the index Raw. The only way to use a `Socket Raw` is to `Bind` it, and the only way to use a `Socket Bound` is to `Connect` it. This forces us to follow a protocol when initializing a `Socket`.

3 $\mathcal{H}$: A LANGUAGE WITH BOTH DATA AND CODATA

We have seen how codata can be used in many of the cases where object-oriented programming is also useful. In fact, several of our examples come from object-oriented languages. To begin to formalize the similarities of codata and objects, we define the language $\mathcal{H}$ which contains two sorts of types: data and codata. Though we have not discussed them much, data types have been included for two reasons. At first glance data type constructors may look similar to object constructors specified in a class declaration; we aim to show how they are different. Secondly, data types have long been missing from object-oriented languages and we hope to show how easily they can be added.

For notation, we use two representations of a list for convenience. The notation $\overline{S}$ is a list containing the elements S. We write $S_0, S_1, \ldots_n$ to mean a list of n elements that we wish to index in to; we omit the subscript n when we do not care about the list's length. For both data constructors and codata observations, we omit the parentheses when they take no arguments, *e.g.* $e.\mathsf{fst}$ instead of $e.\mathsf{fst}()$.

This presentation differs from other languages combining data and codata [2, 9, 22] in that we have no special function type, instead it is added as a codata type in the same way object-oriented programmers implement functions as an interface with a single method. For convenience, we embed the λ-calculus in $\mathcal{H}$ as the following macro:

$$[\![x]\!] = x \qquad [\![t\ e]\!] = [\![t]\!].\mathsf{app}([\![e]\!]) \qquad [\![\lambda x.u]\!] = \{\mathsf{app}(x) \to [\![u]\!]\}$$

For example, the function call $(\lambda x.1 + x)\ 41$ is represented as $\{\mathsf{app}(x) \to x + 1\}.\mathsf{app}(41)$.

3.1 $\mathcal{H}$ Syntax

Figure 1 gives the syntax of $\mathcal{H}$. Types are divided into either data or codata types, where types like "bool" and "nat" are data and "fun" (function types) and "stream" are codata. Since we have no builtin types, every type must be specified by a (co)data declaration. A data type declaration X is specified by a set of constructors with the form $\mathsf{K}_i : \overline{\tau}_i \to \mathsf{X}$, where K_i is the constructor's name, $\overline{\tau}_i$ is the list of arguments for the constructor, and the resulting type of X. On the other hand, a codata declaration A is specified by a set of observations with the form $\mathsf{O}_i : \mathsf{A} \to \overline{\tau}_i \to \rho_i$, where O_i is the observation's name, $\overline{\tau}_i$ is a list of arguments that the observation requires, and ρ_i is the final

Variables

$$\mathsf{X},\mathsf{Y},\mathsf{Z} \in \mathit{Data} \qquad \mathsf{A},\mathsf{B},\mathsf{C} \in \mathit{Codata} \qquad x,y,z \in \mathit{Term}$$

Types

$$\begin{array}{llcll} \tau,\rho,\varphi,\gamma \in & \mathit{Type} & ::= & \mathsf{X} & \text{Data} \\ & & \mid & \mathsf{A} & \text{Codata} \\ \\ & \mathit{Declaration} & ::= & \textbf{data}\ \mathsf{X}\ \textbf{where}\ \{\overline{\mathsf{K}_i : \overline{\tau}_i \to \mathsf{X}}\} & \text{Data decl.} \\ & & \mid & \textbf{codata}\ \mathsf{A}\ \textbf{where}\ \{\overline{\mathsf{O}_i : \mathsf{A} \to \overline{\tau}_i \to \rho_i}\} & \text{Codata decl.} \end{array}$$

Terms

$$\begin{array}{llcll} e,t,u \in & \mathit{Term} & ::= & x & \text{Variable} \\ & & \mid & \textbf{fix}\ x{:}\tau\ \textbf{in}\ e & \text{Fixed point} \\ & & \mid & \mathsf{K}(\overline{e}) & \text{Data intro.} \\ & & \mid & \textbf{case}\ e\ \textbf{of}\ \{\overline{\mathsf{K}_i(\overline{x}_i) \to t_i}\} & \text{Data elim.} \\ & & \mid & \{\overline{\mathsf{O}_i(\overline{x}_i) \to e_i}\} & \text{Codata intro.} \\ & & \mid & e.\mathsf{O}(\overline{t}) & \text{Codata elim.} \end{array}$$

Fig. 1. Syntax of $\mathcal{H}$

Terms $\boxed{\Gamma \vdash t : \tau}$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \textbf{fix}\ x{:}\tau\ \textbf{in}\ e : \tau}$$

$$\frac{\Gamma \vdash e_0 : \tau_0 \quad \ldots}{\Gamma, \mathsf{K} : \tau_0 \cdots \to \mathsf{X} \vdash \mathsf{K}(e_0,\ldots) : \mathsf{X}} \qquad \frac{\Gamma \vdash e : \mathsf{X} \quad \mathsf{K}_0 : \overline{\rho}_0 \to \mathsf{X} \in \Gamma \quad \Gamma, \overline{x_0 : \rho_0} \vdash t_0 : \tau \quad \ldots}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ \{\mathsf{K}_0(\overline{x}_0) \to t_0, \ldots\} : \tau}$$

$$\frac{\mathsf{O}_0 : \mathsf{A} \to \overline{\tau}_0 \to \rho_0 \in \Gamma \quad \Gamma, \overline{x_0 : \tau_0} \vdash e_0 : \rho_0 \quad \ldots}{\Gamma \vdash \{\mathsf{O}_0(\overline{x}_0) \to e_0, \ldots\} : \mathsf{A}} \qquad \frac{\Gamma \vdash e : \mathsf{A} \quad \Gamma \vdash t_0 : \rho_0 \quad \ldots}{\Gamma, \mathsf{O} : \mathsf{A} \to \rho_0 \cdots \to \tau \vdash e.\mathsf{O}(t_0,\ldots) : \tau}$$

Fig. 2. Type System of $\mathcal{H}$

type. Notice that every constructor in a data declaration X always produces the same output type X. Dually, every observation in a codata declaration A always requires the same first input type A.

The term level has standard syntax for variables x and self reference (fixed points). In a manner familiar to functional programmers, data types are introduced by fully applying constructors $\mathsf{K}(\overline{e})$ and are eliminated through case expressions which contain branches that pattern match on constructors. In a manner familiar to object-oriented programmers, codata types are introduced by a set of observation declarations (like method declarations) and are eliminated by selecting an observation and supplying its arguments $e.\mathsf{O}(\overline{t})$ (like method invocation). For simplicity, we assume that case expressions have a branch for every constructor of the data type that it is eliminating and sets of observations have a branch for every observation of the codata type that it is introducing.

3.2 $\mathcal{H}$ Static Semantics

The type system for $\mathcal{H}$ is given in Figure 2. It is composed of a single judgement for terms $\boxed{\Gamma \vdash t : \tau}$ which assumes that the declared constructors and observations are found in the environment Γ. The rules for variables, fixed points, data introduction and elimination are standard.

Contexts and Values

$$
\begin{array}{rrcl}
E \in & \mathit{Context} & ::= & \Box \\
& & \mid & \textbf{fix}\ x{:}\tau\ \textbf{in}\ E \\
& & \mid & \mathsf{K}(\overline{V}, E, \overline{t}) \\
& & \mid & \textbf{case}\ E\ \textbf{of}\ \{\overline{\mathsf{K}_i(\overline{x}_i) \to t_i}\} \\
& & \mid & E.\mathsf{O}(\overline{t}) \\
& & \mid & V.\mathsf{O}(\overline{V}, E, \overline{t})
\end{array}
\qquad
\begin{array}{rrcl}
V \in & \mathit{Value} & ::= & \mathsf{K}(\overline{V}) \\
& & \mid & \{\overline{\mathsf{O}_i(\overline{x}_i) \to e_i}\}
\end{array}
$$

Reduction Rules $\boxed{t \mapsto e}$

$$
\begin{array}{rcl}
\textbf{fix}\ x{:}\tau\ \textbf{in}\ e & \mapsto & e[\textbf{fix}\ x{:}\tau\ \textbf{in}\ e/x] \\
\textbf{case}\ \mathsf{K}_i(\overline{V})\ \textbf{of}\ \{\ldots, \mathsf{K}_i(\overline{x}_i) \to e_i, \ldots\} & \mapsto & e_i[\overline{V_i/x_i}] \\
\{\ldots, \mathsf{O}_i(\overline{x}_i) \to e_i, \ldots\}.\mathsf{O}_i(\overline{V}) & \mapsto & e_i[\overline{V_i/x_i}]
\end{array}
$$

Fig. 3. Call-by-value Semantics for $\mathcal{H}$

The rules for codata introduction and elimination differ from most other presentations of codata [3, 9, 22] because we include arguments in observations. Codata introduction checks that all of the observations conform to the interface given in the codata declaration, *i.e.* they take the correct number and type arguments, and produce a result of the correct type. The inclusion of arguments is what allows us to encode the λ-calculus as codata. The rule checking that each branch of a codata introduction is similar to the standard typing rule for lambdas. That is, we check that the term in the branch has the type ρ_i given an environment extended with the observations arguments. This alludes to the idea that codata and objects are simply multi-entry functions [16]. And like the standard function elimination rule (*i.e.* function application), codata elimination checks that the codata and the arguments match the types given in the observation's declaration.

3.3 $\mathcal{H}$ Dynamic Semantics

We give the call-by-value operational semantics for $\mathcal{H}$ in Figure 3. A call-by-name semantics for codata is given in [9] but we omit it here because call-by-value is used in the majority of object-oriented languages. The dynamic semantics includes evaluation contexts, values, and reduction rules. Only data whose arguments are fully evaluated are treated as values, whereas every codata introduction is a value. To produce data type values, we evaluate all of the arguments of an applied constructor from left to right before returning a value. On the other hand, codata introductions are treated as values immediately which gives them the on-demand semantics discussed in Section 2.2, even in a call-by-value language. For evaluation contexts, the fixed point, constructor, and case expressions contexts are all standard. The contexts for observation application ($E.\mathsf{O}(\overline{u})$ and $V.\mathsf{O}(\overline{V}, E, \overline{t})$) correspond to the function application contexts ($E\ e$ and $V\ E$) in the call-by-value λ-calculus, where the codata object must first be evaluated to a value then all of the arguments are evaluated from left to right.

There are only three reduction rules. The fixed point rule is standard. The case expression rule selects the branch matching constructor used while substituting arguments for variables. Similarly, the observation rule selects the branch matching the observation used while substituting arguments for variables.

Subtypes $\boxed{\tau <: \varphi}$

$$\frac{}{\tau <: \tau}\ \textit{Refl} \qquad \frac{\tau <: \rho \quad \rho <: \varphi}{\tau <: \varphi}\ \textit{Trans}$$

$$\frac{\begin{array}{l}(\textbf{codata}\ \mathsf{A}\ \textbf{where}\ \{\mathsf{O}_0 : \mathsf{A} \to \overline{\tau_0} \to \rho_0, \ldots_{n+m}\})[\mathsf{B}/\mathsf{A}] \\ \ \textbf{codata}\ \mathsf{B}\ \textbf{where}\ \{\mathsf{O}_0 : \mathsf{B} \to \overline{\tau_0} \to \rho_0, \ldots_{n}\}\end{array}}{A <: B}\ \textit{Breadth}$$

$$\frac{\begin{array}{l}\textbf{codata}\ \mathsf{A}\ \textbf{where}\ \{\mathsf{O}_0 : \mathsf{A} \to \overline{\rho_0} \to \tau_0, \ldots_{n}\} \\ \textbf{codata}\ \mathsf{B}\ \textbf{where}\ \{\mathsf{O}_0 : \mathsf{B} \to \overline{\varphi_0} \to \gamma_0, \ldots_{n}\} \\ \overline{\varphi_0 <: \rho_0[\mathsf{B}/\mathsf{A}]} \quad \tau_0[\mathsf{B}/\mathsf{A}] <: \gamma_0 \quad \ldots_n\end{array}}{A <: B}\ \textit{Depth}$$

Terms $\boxed{\Gamma \vdash t : \tau}$

$$\ldots \qquad \frac{\Gamma \vdash t : \varphi \quad \varphi <: \tau}{\Gamma \vdash t : \tau}\ \textit{Sub}$$

Fig. 4. Subtyping rules for $\mathcal{H}_{<:}$

3.4 Properties

$\mathcal{H}$ enjoys the type safety properties of progress and preservation. That is, every well-typed term is either a value of a data or codata type or there is a step it can take. Progress holds over closed terms. The proofs are provided in Appendix A.

Theorem 1 ($\mathcal{H}$ Progress). *If* $\vdash t : \tau$*, then either* t *is a value or* $t \mapsto t'$.

Theorem 2 ($\mathcal{H}$ Preservation). *If* $\Gamma \vdash t : \tau$*, and* $t \mapsto t'$*, then* $\Gamma \vdash t' : \tau$.

4 $\mathcal{H}_{<:}$: CODATA WITH SUBTYPING

The next step in route to comparing codata to objects is to incorporate one of the defining features of object-orientation: subtyping. Intuitively, the notion of subtyping is the exact same for codata as it is for objects and records. That is, if a codata type A has the same observations as another codata type B plus some more, then we can safely use a term of type A where a term of type B is required. For example, a triple codata type with the observations fst : Triple → int, snd : Triple → int, and thd : Triple → int can be used in place of a pair codata type that only contains the first two observations.

Adding subtyping to $\mathcal{H}$ requires that we make changes only to the type system (see Figure 4). We call the new language $\mathcal{H}_{<:}$. First, we define a subtyping relation $\boxed{\tau <: \sigma}$ that states "τ is a subtype of σ". The first two rules state that (<:) is both reflexive and transitive. Next, we define two ways that a codata type can be a subtype of another. The first states that if a codata has all of the *same* observations and more of another, then it is a subtype of the other; this is known as width or breadth subtyping. For instance, a codata type ⊤ with no observations (**codata** ⊤ **where** {}) is a supertype of all codata types with one or more observations; this is similar to the `Object` type in Java. The second subtyping rule fore codata states that if the argument and return types of a codata's observations are subtypes of another codata's, then it is a subtype of the other; this is known as depth subtyping. In the depth subtyping rule, the subtyping relation for arguments is

Variables

$$\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E} \in \textit{Class name/Type}$$

Programs

$\mathcal{D} \in$	*Class declaration*	$::=$	$\textbf{class}\ \mathsf{A}\ \textbf{extends}\ \mathsf{B}\ \{\overline{\mathsf{C}\ f};\ \mathcal{K};\ \overline{\mathcal{M}}\}$	
$\mathcal{K} \in$	*Constructor declaration*	$::=$	$\mathsf{A}(\overline{\mathsf{C}\ f})\ \{\textbf{super}(\overline{f});\ \overline{\textbf{this}.f = f}\}$	
$\mathcal{M} \in$	*Method declaration*	$::=$	$\mathsf{A}\ \mathsf{m}(\overline{\mathsf{B}\ x})\ \{\textbf{return}\ t;\}$	
$t, e, u \in$	*Terms*	$::=$	x	Variable
		$\mid$	$t.f$	Field access
		$\mid$	$t.\mathsf{m}(\overline{e})$	Method invocation
		$\mid$	$\textbf{new}\ \mathsf{A}(\overline{t})$	Object creation
		$\mid$	$(\mathsf{A})\ t$	Cast

Fig. 5. Syntax of Featherweight Java

contravariant, while results are covariant. This corresponds to the function subtype rule in the λ-calculus with subtypes where functions are contravariant in their domain and covariant in their codomain.

In both the breadth and depth rules, the subtyping relation for codata is structural because we substitute the names of one codata type for another. This structural subtyping differs from the nominal subtyping found in classed-based object-oriented languages like Java and C#. Structural subtyping subsumes nominal subtyping, but it makes typechecking $\mathcal{H}_{<:}$ not syntax-directed. We will expand on this difference more when we compile classes into codata in Section 6.

The final step in adding subtyping to $\mathcal{H}$ is to extend the term typing rules to include a subsumption rule. This extra rule allows us to replace one type with another if it is a subtype. This can be seen as implicit *upcasting*, which allows us to forget observations that we do not need.

$\mathcal{H}_{<:}$ with its extended type system maintains the same properties of progress and preservation as $\mathcal{H}$. These proofs are in Appendix B.

Theorem 3 ($\mathcal{H}_{<:}$ Progress). *If $\vdash t : \tau$, then either t is a value or $t \mapsto t'$.*

Theorem 4 ($\mathcal{H}_{<:}$ Preservation). *If $\Gamma \vdash t : \tau$ and $t \mapsto t'$, then $\Gamma \vdash t' : \tau$.*

5 FEATHERWEIGHT JAVA: A CLASS-BASED OBJECT-ORIENTED LANGUAGE

Popular object-oriented programming languages focus on the notion of a *class* which can be thought of as a template for constructing and extending objects in addition to specifying their interface. Featherweight Java (FJ) [14] is a standard model of a class-based language that was designed to help model extensions to Java.

5.1 FJ Syntax

The syntax of FJ is given in Figure 5. It is simple, containing only a five productions. The only types in FJ are object types that are either declared or the base/top object type: $\top$. Class declarations introduce new object types. They specify a new class name, its supertype, a set of fields $\mathsf{C}\ f$ and their types, a single constructor $\mathcal{K}$, and a set of methods $\mathcal{M}$. Fields of the class being defined are distinct from fields of the supertype, whereas methods of the class being defined may overload methods of the supertype. Constructors bear the same name as the type created and take only the class's fields and those of its supertype as arguments. The syntax of FJ's constructors suggests two common Java practices: that the super class's constructor is initialized with the relevant fields

Classes $\boxed{\mathcal{D}\ \textbf{OK}}$

$$\frac{\begin{array}{c}\mathcal{K} = \mathsf{A}(\mathsf{D}_0\ g_0, \ldots, \mathsf{E}_0\ h_0, \ldots)\ \{\textbf{super}(g_0, \ldots); \textbf{this}.f_0 = h_0; \ldots\} \\ \mathsf{fields}(\mathsf{B}) = \mathsf{D}_0\ g_0, \ldots \\ \mathcal{M}_0\ \textbf{OK IN}\ \mathsf{A} \quad \ldots\end{array}}{\textbf{class}\ \mathsf{A}\ \textbf{extends}\ \mathsf{B}\ \{\mathsf{C}_0\ f_0, \ldots; \mathcal{K}; \mathcal{M}_0, \ldots\}\ \textbf{OK}}$$

Methods $\boxed{\mathcal{M}\ \textbf{OK IN}\ \mathsf{A}}$

$$\frac{\begin{array}{c}x_0 : \mathsf{B}_0, \ldots, \textbf{this} : \mathsf{C} \vdash e : \mathsf{D} \\ \mathsf{D} <: \mathsf{A} \\ \mathsf{m} : \mathsf{B}_0 \cdots \to \mathsf{A} \in \mathsf{methods}(\mathsf{A}) \cup \mathsf{methods}(\mathsf{super}(\mathsf{A}))\end{array}}{\mathsf{A}\ \mathsf{m}(\mathsf{B}_0\ x_0, \ldots)\ \{\textbf{return}\ e;\}\ \textbf{OK IN}\ \mathsf{C}}$$

Terms $\boxed{\Gamma \vdash t : \tau}$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}$$

$$\frac{\Gamma \vdash e : \mathsf{A} \quad \mathsf{fields}(\mathsf{A}) = \mathsf{C}_0\ f_0, \ldots}{\Gamma \vdash e.f_i : \mathsf{C}_i}$$

$$\frac{\Gamma \vdash e : \mathsf{A} \quad \mathsf{mtype}(\mathsf{m}, \mathsf{A}) = \mathsf{D}_0, \cdots \to \mathsf{C} \qquad \Gamma \vdash u_0 : \mathsf{B}_0 \quad \mathsf{B}_0 <: \mathsf{D}_0 \quad \ldots}{\Gamma \vdash e.\mathsf{m}_i(u_0, \ldots) : \mathsf{C}}$$

$$\frac{\mathsf{fields}(\mathsf{C}) = \mathsf{D}_0\ f_0, \ldots \qquad \Gamma \vdash u_0 : \mathsf{B}_0 \quad \mathsf{B}_0 <: \mathsf{D}_0 \quad \ldots}{\Gamma \vdash \textbf{new}\ \mathsf{C}(u_0, \ldots) : \mathsf{C}}$$

$$\frac{\Gamma \vdash e : \mathsf{D} \quad \mathsf{D} <: \mathsf{C}}{\Gamma \vdash (\mathsf{C})\ e : \mathsf{C}} \quad \frac{\Gamma \vdash e : \mathsf{D} \quad \mathsf{C} <: \mathsf{D} \quad \mathsf{C} \neq \mathsf{D}}{\Gamma \vdash (\mathsf{C})\ e : \mathsf{C}} \quad \frac{\Gamma \vdash e : \mathsf{D} \quad \mathsf{D} \not<: \mathsf{C} \quad \mathsf{C} \not<: \mathsf{D}}{\Gamma \vdash (\mathsf{C})\ e : \mathsf{C}}$$

Fig. 6. Type System of FJ

and that the class's fields are initialized with the arguments of the constructors. Finally, method declarations specify a return type, a method name m, a list of typed arguments, and a term.

Terms in FJ contain some of the same syntax found in $\mathcal{H}_{<:}$: variables and method invocation (which field access is a no argument version of). For working with classes, terms contain syntax for object creation through object constructor application and a new cast operation.

We will make use of several auxiliary functions that require knowledge of all class declarations. fields(A) and methods(A) return all of the fields and methods of class A including those of all of its supertypes. super(A) will return the supertype that A extends and constr(A) returns the constructor declaration for A. Lastly, supertypes(A) returns a list containing a string representation for all of the supertypes of A.

5.2 FJ Static Semantics

The type system for FJ is found in Figure 6. It contains judgements for class declarations, methods, and terms. The class declaration judgement checks the correctness of the object constructor and that the methods are well-formed. An object's constructor is correct iff all of the fields of the supertype constructor are passed to its initialization **super**($\overline{g}$) and if all of the fields for this class have been set with **this**.$f = f$.

Checking that methods are correct is given by a new judgement $\boxed{\mathcal{M}\ \textbf{OK IN}\ \mathsf{A}}$ stating that $\mathcal{M}$ is a well-formed method for the class A. This requires that the method body is well-typed in the

Contexts and Values

$$\begin{array}{rllllrll}
E \in & \mathit{Context} & ::= & \square & \qquad & V, W \in & \mathit{Values} & ::= \ \textbf{new}\ \mathsf{A}(\overline{V}) \\
 & & \mid & E.f & & & & \\
 & & \mid & E.\mathsf{m}(\overline{t}) & & & & \\
 & & \mid & V.\mathsf{m}(\overline{V}, E, \overline{t}) & & & & \\
 & & \mid & \textbf{new}\ \mathsf{A}(\overline{V}, E, \overline{t}) & & & & \\
 & & \mid & (\mathsf{A})\ E & & & &
\end{array}$$

Reduction Rules $\boxed{t \mapsto e}$

$$\begin{array}{rcll}
(\textbf{new}\ \mathsf{C}(\overline{V})).f_i & \mapsto & V_i & \\
(\textbf{new}\ \mathsf{C}(\overline{V})).\mathsf{m}_i(\overline{W}) & \mapsto & u_i[\overline{W/x}][\textbf{new}\ \mathsf{C}(\overline{V})/\textbf{this}] & \\
(\mathsf{A})\ (\textbf{new}\ \mathsf{B}(\overline{V})) & \mapsto & \textbf{new}\ \mathsf{B}(\overline{V}) & \qquad \textbf{where}\ \mathsf{B} <: \mathsf{A}
\end{array}$$

Fig. 7. Featherweight Java Operational Semantics

environment that includes the method arguments and the self-referencing **this** variable. The type of the body must be a subtype of the declared method output type. Finally, we check that the method has the same type declared in either A or one of its super types. This final property is required for method override, ensuring that if we override a supertype's method, then its type remains unchanged.

The judgement for terms is $\boxed{\Gamma \vdash t : \tau}$ and it assumes knowledge of all of the classes used. The rules for field access and method invocation check that the object being observed actually contains the field or method requested. Method invocations and object constructor applications have checks that their arguments are subtypes of the declared argument type. Finally, we look to the three cast rules. We can say that a cast is well-typed if the argument is being cast to a supertype, cast to a subtype, or if it is being cast to an unrelated type. Thus, there is no way to statically rule out any cast. We will see next that casts only serve a purpose at runtime.

5.3 FJ Dynamic Semantics

The operational rules as well as evaluation contexts and values for FJ are given in Figure 7 following the call-by-value semantics given in Pierce [19]. The evaluation contexts resemble those for $\mathcal{H}_{<:}$ closely, where we must evaluate an object to a value before performing a field access, a method call, or a cast. And because the semantics is call-by-value, arguments of method calls and constructors must be fully evaluated. The only value in FJ is an applied object constructor.

As designed, the reduction rules for FJ are simple. Accessing the ith field is simply returning the ith field of the constructor. Invoking the ith method involves jumping to the ith method body of the class doing substitutions for arguments and a special substitution of the current object for the **this** keyword. Finally we see that the casting rule is not so much a computation as it is a runtime check. If the cast is not an upcast, then the program will get stuck.

6 COMPILING FEATHERWEIGHT JAVA TO $\mathcal{H}_{<:}$

Finally, we are prepared to present the relationship between codata and class-based objects as a compilation of objects to codata. First, we present the transformation with examples, then we give the full transformation, and last, we discuss its correctness.

6.1 Compilation By Example

Classes and Constructors. As the only types in Featherweight Java, we must start with how to represent the essence of classes in $\mathcal{H}_{<:}$. Intuitively, classes are transformed into codata declarations which specify their interface. Constructors are treated specially because they are not part of this interface, *i.e.* we cannot call a constructor as a method of an object. So constructors are compiled to functions that introduce instances of the new codata type. As an example, let's consider this notion of a product type in the FJ given in [14]:

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Pair}\ \textbf{extends}\ \top\ \{\\
\quad \top\ \mathit{fst};\\
\quad \top\ \mathit{snd};\\
\quad \mathsf{Pair}(\top\ \mathsf{fst}, \top\ \mathsf{snd})\ \{\textbf{super}(); \textbf{this}.\mathit{fst} = \mathit{fst}; \textbf{this}.\mathit{snd} = \mathit{snd};\}\\
\quad \mathsf{setfst}(\top\ \mathsf{newfst})\ \{\textbf{return}\ (\textbf{new}\ \mathsf{Pair}(\mathit{newfst}, \textbf{this}.\mathit{snd});)\}\\
\}
\end{array}
$$

Given any two objects we can construct a Pair object with the fields "fst" and "snd". There is also a "setfst" method which will return a new Pair object with a different first component[2].

We translate this class into $\mathcal{H}_{<:}$ by creating a codata declaration Pair that captures the class's interface. For the constructor, we define a function *newPair* (making use of the encoding of the λ-calculus given in Section 3).

$$
\begin{array}{l}
\textbf{codata}\ \mathsf{Pair}\ \textbf{where}\\
\quad \begin{array}{lcl}
\mathsf{fst} & : & \mathsf{Pair} \to \top\\
\mathsf{snd} & : & \mathsf{Pair} \to \top\\
\mathsf{setfst} & : & \mathsf{Pair} \to \top \to \mathsf{Pair}
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathit{newPair} = \textbf{fix}\ \mathit{newPair} : \top \to \top \to \mathsf{Pair}\ \textbf{in}\\
\qquad\qquad \lambda \mathit{fst}{:}\top.\lambda \mathit{snd}{:}\top.\\
\qquad\qquad \left\{ \begin{array}{ccl}
\mathsf{fst} & \to & \mathit{fst}\\
\mathsf{snd} & \to & \mathit{snd}\\
\mathsf{setfst}(\mathit{newfst}) & \to & \mathit{newPair}\ \mathit{newfst}\ \mathit{snd}
\end{array} \right\}
\end{array}
$$

Since both the fields and methods of a class are part of its interface, the codata type declaration generated contains observations for both. The constructor for the class (*newPair*) is a function that takes the fields of a class as arguments before producing the codata type. Using function application for the fields of object construction captures the fact that FJ classes have call-by-value fields and call-by-name methods. Since, observation arguments in $\mathcal{H}_{<:}$ are evaluated ahead of time, the translated fields of the class will be evaluated in the same manner as object constructor application in FJ.

Another aspect of the classes is self-reference. In this example, we see self-referenced fields which are handled by simply accessing the respective variable in the closure for the codata type, *e.g.* $\{\mathsf{fst} \to \mathit{fst}, \ldots\}$. Self-referencing constructor applications and method calls require the fixed point to construct a new object. We see this in the "setfst" branch of the codata introduction.

Inheritance. A useful feature of object-oriented programming not captured in the example above is inheritance (since Pair in inherits nothing from the empty object $\top$). For this, let's consider a

[2] In this example, the $\top$ object is used as a "poor man's" polymorphism allowing us to build a pair out of any object since everything is a subtype of $\top$.

Triple object that extends Pair by one field.

$$\begin{array}{l} \textbf{class}\ \mathsf{Triple}\ \textbf{extends}\ \mathsf{Pair}\ \{ \\ \quad \top\ \mathit{thd}; \\ \quad \mathsf{Triple}(\top\ \mathit{fst}, \top\ \mathit{snd}, \top\ \mathit{thd})\ \{\textbf{super}(\mathrm{fst}, \mathrm{snd}); \textbf{this}.\mathit{thd} = \mathit{thd};\} \\ \} \end{array}$$

The generated codata type captures the extended interface by including *all* of the fields and methods of its supertypes in its declaration's observations.

$$\begin{array}{l} \textbf{codata}\ \mathsf{Triple}\ \textbf{where} \\ \quad \begin{array}{lcl} \mathsf{fst} & : & \mathsf{Triple} \to \top \\ \mathsf{snd} & : & \mathsf{Triple} \to \top \\ \mathsf{thd} & : & \mathsf{Triple} \to \top \\ \mathsf{setfst} & : & \mathsf{Triple} \to \top \to \mathsf{Pair} \end{array} \end{array}$$

$$\begin{array}{l} \mathit{newTriple} = \textbf{fix}\ \mathit{newTriple} : \top \to \top \to \top \to \mathsf{Triple}\ \textbf{in} \\ \qquad\qquad \lambda \mathit{fst}{:}\top.\lambda \mathit{snd}{:}\top.\lambda \mathit{thd}{:}\top \\ \qquad\qquad \left\{ \begin{array}{ccl} \mathsf{fst} & \to & \mathit{fst} \\ \mathsf{snd} & \to & \mathit{snd} \\ \mathsf{thd} & \to & \mathit{thd} \\ \mathsf{setfst}(\mathit{newfst}) & \to & \mathit{newPair}\ \mathit{newfst}\ \mathit{snd} \end{array} \right\} \end{array}$$

Notice that the "setfst" method still returns a Pair. To match the notion of methods enforced by FJ's type system (Figure 6), inherited methods *must* take the same types of arguments and return the same type as declared in the supertype. We see that the body of method is also the same as in the supertype. If the class Triple overrode that method, then we would include that method body instead of the one declared in Pair.

Casting. Type casts are used in Featherweight Java only to forget the extra structure of class. This means that in terminating programs, all casts must be upcasts. As an example, the downcast

$$(\mathsf{Quadruple})\ (\textbf{new}\ \mathsf{Triple}(1, 2, 3))$$

gets stuck because we cannot conjure a "fourth" projection out of thin air. One the other hand, the upcast

$$(\mathsf{Pair})\ (\textbf{new}\ \mathsf{Triple}(1, 2, 3))$$

will reach a value (**new** Triple(1, 2, 3)) with the new type Pair. This is safe because we can forget the "thd" projection of Triple if the rest of the program treats this term as a Pair.

In order to replicate this behavior in our compilation, we need to know subtype information at runtime. This is done by adding a special casts$^{\#}$ observation to every compiled class. This represents all of the types that the object can be safely cast to, *i.e.* its own type and that of its supertypes.

$$\begin{array}{l} \textbf{codata}\ \mathsf{Pair}\ \textbf{where} \\ \quad \begin{array}{lcl} \mathsf{casts}^{\#} & : & \mathsf{Pair} \to [\mathsf{String}] \\ \mathsf{fst} & : & \mathsf{Pair} \to \top \\ \mathsf{snd} & : & \mathsf{Pair} \to \top \\ \mathsf{setfst} & : & \mathsf{Pair} \to \top \to \mathsf{Pair} \end{array} \end{array}$$

For the "Triple" codata type, this operation is defined as $\{\mathsf{casts}^{\#} \to [\text{“Triple”}, \text{“Pair”}, \text{“}\top\text{”}], \ldots\}$. We use this information in a compiled cast operation, such as the following:

$$[\![(\mathsf{Pair})\ (e : \mathsf{Triple})]\!] = \textbf{if}\ \text{“Pair”} \in [\![e]\!].\mathsf{casts}^{\#}\ \textbf{then}\ [\![e]\!]\ \textbf{else}\ \Omega$$

Declarations

$$
\begin{array}{rcl}
\mathcal{D}\llbracket \textbf{class}\ \mathsf{A}\ \textbf{extends}\ \mathsf{B}\ \{\overline{\mathsf{C}\ f};\ \mathcal{K};\ \overline{\mathcal{M}}\} \rrbracket & = & \textbf{codata}\ \mathsf{A}\ \textbf{where} \\
 & & \quad \mathsf{casts}^{\#} : \mathsf{A} \to [\mathsf{String}] \\
 & & \quad \overline{\mathcal{D}_f\llbracket \mathsf{C}\ f \rrbracket \mathsf{A}}, \overline{\mathcal{D}_f\llbracket \mathrm{fields}(\mathsf{B}) \rrbracket \mathsf{A}} \\
 & & \quad \overline{\mathcal{D}_m\llbracket \mathcal{M} \rrbracket \mathsf{A} \cup \mathcal{D}_m\llbracket \mathrm{methods}(\mathsf{B}) \rrbracket \mathsf{A}} \\
\mathcal{D}_f\llbracket \mathsf{C}\ f \rrbracket \mathsf{A} & = & \mathsf{f} : \mathsf{A} \to \mathsf{C} \\
\mathcal{D}_m\llbracket \mathsf{C}\ \mathsf{m}(\overline{\mathsf{B}\ x})\ \{\textbf{return}\ t;\} \rrbracket \mathsf{A} & = & \mathsf{m} : \mathsf{A} \to \overline{\mathsf{B}} \to \mathsf{C}
\end{array}
$$

Constructors

$$
\begin{array}{rcl}
\mathcal{K}\llbracket \mathsf{A}(\overline{\mathsf{C}\ f})\ \{\textbf{super}(\overline{f});\ \overline{\textbf{this}.f = f}\} \rrbracket & = & \textbf{fix}\ \mathit{newA} : \overline{\mathsf{C}} \to \mathsf{A}\ \textbf{in} \\
 & & \quad \lambda \overline{f{:}\mathsf{C}}. \\
 & & \quad \left\{ \begin{array}{l}
 \mathsf{casts}^{\#} \to [\text{“A”}] \cup \mathrm{casts}(\mathrm{super}(\mathsf{A})) \\
 \overline{\mathcal{K}_f\llbracket \mathsf{C}\ f \rrbracket}, \overline{\mathcal{K}_f\llbracket \mathrm{fields}(\mathrm{super}(\mathsf{A})) \rrbracket} \\
 \overline{\mathcal{K}_m\llbracket \mathrm{methods}(\mathsf{A}) \rrbracket \mathsf{A}, \overline{f}} \\
 \overline{\mathcal{K}_m\llbracket \mathrm{methods}(\mathrm{super}(\mathsf{A})) - \mathrm{methods}(\mathsf{A}) \rrbracket \mathsf{A}, \overline{f}}
 \end{array} \right\} \\
\mathcal{K}_f\llbracket \mathsf{C}\ f \rrbracket & = & \mathsf{f} \to f \\
\mathcal{K}_m\llbracket \mathsf{A}\ \mathsf{m}(\overline{\mathsf{B}\ x})\ \{\textbf{return}\ t;\} \rrbracket \mathsf{C}, \overline{f} & = & \mathsf{m}(\overline{x{:}\mathsf{B}}) \to \mathcal{T}\llbracket t \rrbracket \overline{[f_i/\textbf{this}.\mathsf{f}_i]}\ \overline{[(\mathit{newC}\ f_0\ \ldots\ f_n).\mathsf{m}/\textbf{this}.\mathsf{m}]}
\end{array}
$$

Terms

$$
\begin{array}{rcl}
\mathcal{T}\llbracket x \rrbracket & = & x \\
\mathcal{T}\llbracket t.f_i \rrbracket & = & \mathcal{T}\llbracket t \rrbracket.\mathsf{f}_i \\
\mathcal{T}\llbracket t.\mathsf{m}(\overline{e}) \rrbracket & = & \mathcal{T}\llbracket t \rrbracket.\mathsf{m}(\overline{\mathcal{T}\llbracket e \rrbracket}) \\
\mathcal{T}\llbracket \textbf{new}\ \mathsf{A}(\overline{t}) \rrbracket & = & \mathcal{K}\llbracket \mathrm{constr}(\mathsf{A}) \rrbracket\ \mathcal{T}\llbracket t \rrbracket_0\ \ldots\ \mathcal{T}\llbracket t \rrbracket_n \\
\mathcal{T}\llbracket (\mathsf{A})\ t \rrbracket & = & \textbf{let}\ x = \mathcal{T}\llbracket t \rrbracket\ \textbf{in} \\
 & & \quad \textbf{if}\ \text{“A”} \in x.\mathsf{casts}^{\#} \\
 & & \quad \textbf{then}\ x \\
 & & \quad \textbf{else}\ \Omega
\end{array}
$$

Fig. 8. Compiling FJ to $\mathcal{H}_{<:}$

We check that the string representation of the type that we are casting to matches one of those in codata's "casts$^{\#}$" observation. If the it matches one then continue, otherwise loop forever. So like Featherweight Java, the translated program will fail to reach a value if a cast is not an upcast.

6.2 The Compilation

The full encoding of FJ in $\mathcal{H}_{<:}$ is given in Figure 8. It is composed of three different translations: translating class declarations into codata declarations $\mathcal{D}\llbracket - \rrbracket$, translating constructor declarations into constructor functions $\mathcal{K}\llbracket - \rrbracket$, and translating FJ terms into $\mathcal{H}_{<:}$ terms $\mathcal{T}\llbracket - \rrbracket$.

As we just saw, classes are mapped to codata types containing the fields and methods of the class as well as those of its supertypes and the special "casts$^{\#}$" observation. Since in FJ the fields of a class are distinct from those of its supertypes, we simply include all of the fields of the class and supertypes. On the other hand, we support method overriding by only including observations for the *union* of the methods of the class and its supertypes. Finally, the translation uses two sub-translations that ensure the fields and methods appear with the correct form in the codata declaration.

Object constructors are mapped to self-referencing function definitions, which can be thought of as an object's closure. The "casts$^{\#}$" observation is set to a list of strings representing the all of the supertypes of the class. All of an object's fields are set to access the closure's local variable for that field. Methods are mapped directly to observations multiple arguments. To handle method overriding, we include the translation of all of the methods of the class and only the methods of the super class that have not been overridden. The body of the method is translated with the term translation. In the term, we must also substitute self-references (**this**) to access the correct fields and methods. Self-referenced fields are substituted for the fields of the closure, whereas self-referenced method calls are substituted for a new construction of the object using the fixed point.

The term translation of fields and methods is a simple mapping to observation applications in $\mathcal{H}_{<:}$. Constructor application maps to function application (which is a macro for observation application). The interesting case is the cast operation as discussed in Section 6.1. It is important to note that this encoding of casts requires data types for the if-statement and checking equality of strings.

6.3 Correctness

Our translation has three properties: translation of classes preserves subtyping, translation of terms preserves types, and translation preserves evaluation for both terminating and non-terminating programs. The proofs of these properties can be found in Appendix C.

Lemma 5 (Subtype preservation). *If* $\mathsf{A} <: \mathsf{B}$ *in* FJ*, then* $[\![\mathsf{A}]\!] <: [\![\mathsf{B}]\!]$ *in* $\mathcal{H}_{<:}$.

Because Featherweight Java's type system allows downcasts and nonsense casts, type preservation for the compilation comes with a restriction. That is, it is only valid for programs containing only upcasts. We argue that this is not much of a restriction, because programs with only upcasts are the only ones that do not get stuck. We also note that it is these upcasts and our compilation of them that make it essential for us to add subtyping to $\mathcal{H}$.

Theorem 6 (Type preservation). *If* $\Gamma \vdash t : \mathsf{A}$ *where* t *contains only upcasts, then* $[\![\Gamma]\!] \vdash \mathcal{T}[\![t]\!] : [\![\mathsf{A}]\!]$.

Like type preservation, evaluation preservation is not so clean cut. This is because in FJ constructor application **new** $\mathsf{A}(\overline{V})$ is value, which is translated to a function (observation) application in $\mathcal{H}_{<:}$. However, applications in $\mathcal{H}_{<:}$ are not values! The translated constructor application must take steps before it becomes a codata value. Therefore, we have the following statement of evaluation preservation to account for this fact.

Theorem 7 (Evaluation Preservation). *If* $t \mapsto^* V$ *for some* $t, V \in$ FJ*, then* $\mathcal{T}[\![t]\!] \mapsto^* W$ *where* W *is a value in* $\mathcal{H}_{<:}$ *such that* $\mathcal{T}[\![V]\!] \mapsto W$.

7 DISCUSSION

7.1 Enriching Object-orientation

We have shown that many object-oriented features are either contained in (Section 2), can be incorporated in (Section 4), or compiled down to (Section 6) codata. In the interest of sharing information in the other direction, let's see how aspects of codata can be incorporated into object-orientation.

Algebraic Data Types. The absence of algebraic data types has long been a pain point for object-oriented languages. Since many concepts can be represented by algebras, it is necessary for programmers to easily express and manipulate them. For instance, programming language grammars are algebras and a compiler is simply a series of transformations from one algebraic data structure to another. The common solution to this problem for object-oriented programmers is to either use the visitor pattern or type cases to encode the algebraic structure of interest. However, these solutions include either verbose boilerplate or incomplete runtime type checking, respectively. The lack of the ability to express these structures directly can easily lead to programmer error. In addition to reducing the chance for programmer error created by these encodings, the inclusion of language support for algebraic data types allows compilers to statically check whether all cases are covered. These completeness checks are not possible with encodings of algebras because the compiler does not know all the cases. Luckily, languages related to linear logic via the Curry-Howard correspondence (like $\mathcal{H}$) can show us how to integrate data and codata types.

Fully Corecursive Objects. Corecursive definitions specify how to construct the *next* part of a self-similar structure, whereas recursive definitions specify what to do with a smaller part of a self-similar structure. The former allows for specification of possibly infinite objects. These can occur often in practice; for instance, a database server is a process that is meant to run forever. In an attempt to include corecursion in an object-oriented setting, Ancona and Zucca [5] extended Featherweight Java with *regular* corecursive objects. A similar extension was added to the functional language Ocaml [15]. Regular corecursive objects are restricted in that they can only represent cyclic structures with a finite representation, *i.e.* an infinite stream of 0s.

Fortunately, $\mathcal{H}$'s codata types already support unrestricted corecursion. A key aspect of codata is that it is only evaluated *on-demand* regardless of the strategy of the host language, as discussed in Section 2.2. It is this aspect of codata that allows us to define exactly how to build up an infinite structure, but not run into an infinite loop trying to construct it. The client of the codata will decide how much of the structure is generated. Interestingly, Featherweight Java can already support this sort of unrestricted corecursive structures because it has methods which are evaluated on-demand. There is only an issue with corecursion if the programmer tries to include the corecursion as a call-by-value field.

Introducing Objects with Copatterns. Ancona and Zucca [5] argue that Featherweight Java does not have a good abstraction for specifying corecursive objects. Another advantage of using codata as the base of objects is that it allows us to use copattern matching to define them [2]. Dual to pattern matching which allows the programmer to look deeply into a nested data structure, copattern matching allows the programmer to specify how to generate several layers of a nested (possibly infinite) codata structure. The poster-child of using copattern matching to define complex corecursive structures is the infinite stream of fibonacci numbers as seen below.

$$\begin{array}{l}\textbf{fix}\ \mathit{fib} : \mathsf{stream}\ \textbf{in}\\ \quad\left\{\begin{array}{lcl}.\mathsf{head} & \to & 1\\ .\mathsf{tail}.\mathsf{head} & \to & 1\\ .\mathsf{tail}.\mathsf{tail} & \to & \mathit{zipwith}\ (+)\ \mathit{fib}\ \mathit{fib}.\mathsf{tail}\end{array}\right\}\end{array}$$

7.2 Conclusion

In exploring the relationship between codata and objects, we have found that codata is a good fit for directly modelling several aspects of object-oriented programming including on-demand evaluation and abstraction. We also saw that by adding subtyping to codata types, we can compile of the notion of classes into codata. Though our compilation is flexible enough to represent the features of Featherweight Java, modern object-oriented languages are large beasts that incorporate many language features not included in it, such as runtime reflection of types and dynamic loading of classes. For future work, we would like to explore how these may interact with our compilation.

In the spirit of the Curry-Howard correspondence, our work suggests that we can exploit the close relationship between codata and objects to expand the expressiveness of object-oriented languages. Additionally, since data and codata integrate so well together, our approach can be used as a two-paradigm intermediate language that suitable for both functional and object-oriented languages.

REFERENCES

[1] Martín Abadi and Luca Cardelli. A theory of primitive objects - scond-order systems. In *Programming Languages and Systems - ESOP'94, 5th European Symposium on Programming, Edinburgh, UK, April 11-13, 1994, Proceedings*, pages 1–25, 1994.

[2] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: Programming infinite structures by observations. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 27–38, 2013.

[3] Andreas M. Abel and Brigitte Pientka. Wellfounded recursion with copatterns: A unified approach to termination and productivity. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 185–196, 2013.

[4] Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 233–249, 2014.

[5] Davide Ancona and Elena Zucca. Corecursive featherweight java. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs, FTfJP 2012, Beijing, China, June 12, 2012*, pages 3–10, 2012.

[6] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[7] William R. Cook. On understanding data abstraction, revisited. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 557–572, 2009.

[8] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, pages 465–490, 2004.

[9] Paul Downen, Zachary Sullivan, Zena M. Ariola, and Simon L. Peyton Jones. Codata in action. In *Proceedings of the 28th European Symposium on Programming*, 2019.

[10] M. Dummett. The logical basis of methaphysics. In *The William James Lectures, 1976*. Harvard University Press, Cambridge, Massachusetts, 1991.

[11] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

[12] John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[13] R. J. M. Hughes. Super-combinators: a new implementation method for applicative languages. In *Proc. ACM Symposium on Lisp and Functional Programming*, pages 1–10, 1982.

[14] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherwieght java: A minimal core calculus for java and GJ. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, USA, November 1-5, 1999.*, pages 132–146, 1999.

[15] Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. Cocaml: Functional programming with regular coinductive types. *Fundam. Inform.*, 150(3-4):347–377, 2017.

[16] Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. 2007.

[17] Paul Blain Levy. *Call-By-Push-Value*. PhD thesis, Queen Mary and Westfield College, University of London, 2001.

[18] Guillaume Munch-Maccagnoni. *Syntax and Models of a non-Associative Composition of Programs and Proofs*. PhD thesis, Université Paris Diderot, 2013.

[19] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[20] Yann Regis-Gianas and Paul Laforgue. Copattern-matchings and first-class observations in ocaml, with a macro. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*, PPDP '17, 2017.

[21] John C. Reynolds. *User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction*, pages 309–317. Springer, 1978.

[22] David Thibodeau. Programming infinite structures using copatterns. Master's thesis, 2015.

[23] David Thibodeau, Andrew Cave, and Brigitte Pientka. Indexed codata types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pages 351–363, 2016.

[24] Philip Wadler. Linear types can change the world! In *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, 1990.

[25] Stephanie Weirich. Depending on types. In *Proceedings of the 19rd ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, 2014.

[26] Noam Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, Carnegie Mellon University, 2009.

A TYPE SAFETY FOR $\mathcal{H}$

Lemma 8 (Inversion of Typing for $\mathcal{H}$). *We have a number of observations based on the conclusion of the typing rules.*

- $\Gamma \vdash x : \tau$ *implies* $x : \tau \in \Gamma$
- $\Gamma \vdash \textbf{fix}\ x{:}\tau\ \textbf{in}\ e : \tau$ *implies* $\Gamma, x : \tau \vdash e : \tau$
- $\Gamma \vdash \{\mathsf{O}_0(\overline{x}_0) \to e_0, \dots\} : \tau$ *implies* $\tau = \mathsf{A}, \mathsf{O}_0 : \mathsf{A} \to \overline{\rho}_0 \to \varphi_0 \in \Gamma)$, *and* $\Gamma, \overline{x_0 : \rho_0} \vdash e_0 : \varphi_0, \dots$
- $\Gamma \vdash e.\mathsf{O}(t_0, \dots) : \tau$ *implies* $\mathsf{O} : \mathsf{A} \to \rho_0 \cdots \to \tau \in \Gamma, \Gamma \vdash e : \mathsf{A}$, *and* $\Gamma \vdash t_0 : \rho_0, \dots$
- $\Gamma \vdash \mathsf{K}(e_0, \dots) : \tau$ *implies* $\tau = \mathsf{X}, \mathsf{K} : \rho_0 \cdots \to \mathsf{X} \in \Gamma$, *and* $\Gamma \vdash e_0 : \rho_0, \dots$.
- $\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ \{\mathsf{K}_0(\overline{x}_0) \to t_0, \dots\} : \tau$ *implies* $\Gamma \vdash e : \mathsf{X}$ *and* $(\mathsf{K}_0 : \overline{\rho}_0 \to \mathsf{X} \in \Gamma) \wedge (\Gamma, \overline{x_0 : \rho_0} \vdash t_0 : \tau)), \dots$

Lemma 9 ($\mathcal{H}$ Canonical Forms). *For $e \in \mathcal{H}$ where e is a value,*

- *If $\Gamma \vdash e : \mathsf{A}$, then e has the form $\{\overline{\mathsf{O}_i(\overline{x}_i) \to t_i}\}$.*
- *If $\Gamma \vdash e : \mathsf{X}$, then e has the form $\mathsf{K}(\overline{V})$.*

Theorem 10 ($\mathcal{H}$ Progress). *If $t \in \mathcal{H}$ and $\vdash t : \tau$, then either t a value or $t \mapsto t'$.*

Proof. By induction on the derivation $\vdash t : \tau$.

- case $x : \tau \vdash x : \tau$ holds vacuously.
- case $\vdash \textbf{fix}\ x{:}\tau\ \textbf{in}\ e : \tau$, $\textbf{fix}\ x{:}\tau\ \textbf{in}\ e \mapsto e[\textbf{fix}\ x{:}\tau\ \textbf{in}\ e/x]$.
- case $\vdash \{\overline{\mathsf{O}_i(\overline{x}_i) \to e_i}\} : \mathsf{A}$, $\{\overline{\mathsf{O}_i(\overline{x}_i) \to e_i}\}$ is a value.
- case $\mathsf{O} : \mathsf{A} \to \rho_0 \cdots \to \tau \vdash e.\mathsf{O}(t_0, \dots) : \tau$,
  By inversion on the typing rule, $\Gamma \vdash e : \mathsf{A}$ and $\Gamma \vdash t_0 : \rho_0, \dots$.
  By the inductive hypothesis, e is a value or $e \mapsto e'$.
  - case e is a value,
    By Lemma 9, e has the form $\{\overline{\mathsf{O}_i(\overline{x}_i) \to u_i}\}$.
    By the inductive hypothesis, $\forall t_i \in t_0, \dots$, either t_i is a value or $t_i \mapsto t_i'$.
    - If all t_i are values, then $\{\dots, \mathsf{O}_i(\overline{x}_i) \to u_i, \dots\}.\mathsf{O}_i(\overline{t}_i) \mapsto u_i\overline{[t_i/x_i]}$
    - Otherwise for the first t_i where $t_i \mapsto t_i'$, we know $\{\overline{\mathsf{O}_i(\overline{x}_i) \to u_i}\}.\mathsf{O}_i(\overline{V}, t_i, \overline{t}) \mapsto \{\overline{\mathsf{O}_i(\overline{x}_i) \to u_i}\}.\mathsf{O}_i(\overline{V}, t_i', \overline{t})$ because $V.\mathsf{O}_i(\overline{V}, E, \overline{t})$ is an evaluation context.
  - case $e \mapsto e'$, then $e.\mathsf{O}(t_0, \dots) \mapsto e'.\mathsf{O}(t_0, \dots)$ because $E.\mathsf{O}(t_0, \dots)$ is an evaluation context.
- case $\vdash \mathsf{K}(\overline{e}) : \mathsf{D}$,
  By the inductive hypothesis, $\forall e_i {\in} \overline{e}$, either e_i is a value or $e_i \mapsto e_i'$.
  - If all e_i are values, then $\mathsf{K}(\overline{e})$ is a value.
  - Otherwise for the first e_i where $e_i \mapsto e_i'$, we know $\mathsf{K}(\overline{V}, e_i, \overline{e}) \mapsto \mathsf{K}(\overline{V}, e_i', \overline{e})$ because $\mathsf{K}(\overline{V}, E, \overline{e})$ is an evaluation context.
- case $\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ \{\overline{\mathsf{K}_i(\overline{x}_i) \to t_i}\} : \tau$,
  By inversion on the typing rule, $\Gamma \vdash e : \mathsf{X}$ and $\mathsf{K}_0 : \overline{\rho}_0 \to \mathsf{X} \in \Gamma$ and $\Gamma, \overline{x_i : \rho_0} \vdash t_0 : \tau, \dots$.
  By the inductive hypothesis, e is either a value or $e \mapsto e'$,
  - case e is a value,
    By Lemma 9, e has the form $\mathsf{K}_i(\overline{V})$.
    Thus, $\textbf{case}\ \mathsf{K}_i(\overline{V})\ \textbf{of}\ \{\dots, \mathsf{K}_i(\overline{x}_i) \to t_i, \dots\} \mapsto t_i\overline{[V_i/x_i]}$.
  - case $e \mapsto e'$, then $\textbf{case}\ e\ \textbf{of}\ \{\overline{\mathsf{K}_i(\overline{x}_i) \to t_i}\} \mapsto \textbf{case}\ e'\ \textbf{of}\ \{\overline{\mathsf{K}_i(\overline{x}_i) \to t_i}\}$ because we have $\textbf{case}\ E\ \textbf{of}\ \{\overline{\mathsf{K}_i(\overline{x}_i) \to t_i}\}$ as an evaluation context.

□

Lemma 11 ($\mathcal{H}$ Substitution). *If $\Gamma, x : \tau \vdash t : \rho$ and $\Gamma \vdash e : \tau$, then $\Gamma \vdash t[e/x] : \rho$.*

Theorem 12 ($\mathcal{H}$ Preservation). *If $t \in \mathcal{H}, \Gamma \vdash t : \tau$, and $t \mapsto t'$, then $\Gamma \vdash t' : \tau$.*

Proof. By induction on the derivation $\Gamma \vdash t : \tau$.

- case $\Gamma, x : \tau \vdash x : \tau$ is immediate, since there is no $t \mapsto t'$.
- case $\Gamma \vdash \mathbf{fix}\ x{:}\tau\ \mathbf{in}\ e : \tau$,
  There is one reduction for t, that is $\mathbf{fix}\ x{:}\tau\ \mathbf{in}\ e \mapsto e[\mathbf{fix}\ x{:}\tau\ \mathbf{in}\ e/x]$.
  By inversion, we know $\Gamma, x : \tau \vdash e : \tau$.
  By Lemma 11, $\Gamma \vdash e[\mathbf{fix}\ x{:}\tau\ \mathbf{in}\ e/x] : \tau$.
- case $\Gamma \vdash \{\overline{\mathsf{O}_i(\overline{x}_i) \to e_i}\} : \tau$ is immediate, since there is no $t \mapsto t'$.
- case $\Gamma, \mathsf{O} : \mathsf{A} \to \overline{\rho} \to \tau \vdash e.\mathsf{O}(\overline{t}) : \tau$,
  By inversion on the typing rule, we know $\Gamma \vdash e : \mathsf{A}$ and $\overline{\Gamma \vdash t : \rho}$.
  There are three reductions to consider:
  - case $e.\mathsf{O}(\overline{t}) \mapsto e'.\mathsf{O}(\overline{t})$, we know $\Gamma \vdash e'.\mathsf{O}(\overline{t}) : \tau$ by the inductive hypothesis and the observation typing rule.
  - case $V.\mathsf{O}(\overline{V}, t_i, \overline{t}) \mapsto V.\mathsf{O}(\overline{V}, t_i', \overline{t})$, we know $\Gamma \vdash V.\mathsf{O}(\overline{V}, t_i', \overline{t}) : \tau$ by the inductive hypothesis and the observation typing rule.
  - case $\{\dots, \mathsf{O}_i(\overline{x}_i) \to u_i, \dots\}.\mathsf{O}_i(\overline{V}_i) \mapsto u_i[\overline{V_i/x_i}]$,
    By inversion on the typing rule for e, *i.e.* $\Gamma \vdash \{\dots, \mathsf{O}_i(\overline{x}_i) \to u_i, \dots\} : \mathsf{A}$, we know that $\Gamma, \overline{x_i : \rho_i} \vdash u_i : \tau$.
    By Lemma 11, $\Gamma \vdash u_i[\overline{V_i/x_i}] : \tau$.
- case $\Gamma, \mathsf{K} : \overline{\tau} \dots \to \mathsf{X} \vdash \mathsf{K}(\overline{e}) : \mathsf{X}$,
  By inversion on the typing rule, we know $\overline{\Gamma \vdash e : \tau}$.
  There is one reduction rule to consider, $\mathsf{K}(\overline{V}, e_i, \overline{e}) \mapsto \mathsf{K}(\overline{V}, e_i', \overline{e})$.
  By the inductive hypothesis and the constructor typing rule, we know $\Gamma \vdash \mathsf{K}(\overline{V}, e_i', \overline{e}) : \mathsf{X}$.
- case $\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ \{\overline{\mathsf{K}_i(\overline{x}_i) \to t_i}\} : \tau$,
  By inversion on the typing rule, we know $\Gamma \vdash e : \mathsf{X}$ and $\mathsf{K}_0(\overline{x}_0) : \overline{\rho}_0 \to \mathsf{X}$ and $\Gamma, \overline{x_0 : \rho_0} \vdash t_0 : \tau, \dots$.
  There are two reductions to consider:
  - case $\mathbf{case}\ e\ \mathbf{of}\ \{\overline{\mathsf{K}_i(\overline{x}_i) \to t_i}\} \mapsto \mathbf{case}\ e'\ \mathbf{of}\ \{\overline{\mathsf{K}_i(\overline{x}_i) \to t_i}\}$, we know $\Gamma \vdash \mathbf{case}\ e'\ \mathbf{of}\ \{\overline{\mathsf{K}_i(\overline{x}_i) \to t_i} : \tau$ by the inductive hypothesis and the case-expression typing rule.
  - case $\mathbf{case}\ \mathsf{K}_i(\overline{V})\ \mathbf{of}\ \{\dots, \mathsf{K}_i(\overline{x}_i) \to t_i, \dots\} \mapsto t_i[\overline{V_i/x_i}]$,
    By inversion on the typing rule for e, *i.e.* $\Gamma, \mathsf{K}_i : \overline{\rho}_i \to \mathsf{X} \vdash \mathsf{K}_i(\overline{V}) : \mathsf{X}$, we know that $\overline{\Gamma \vdash V_i : \rho_i}$.
    By Lemma 11, we know $\Gamma \vdash t_i[\overline{V_i/x_i}] : \tau$.

□

B TYPE SAFETY FOR $\mathcal{H}_{<:}$

Lemma 13 (Inversion of Subtyping for codata in $\mathcal{H}_{<:}$). *If $A <: B$, then we have the codata declarations* $(\mathbf{codata}\ \mathsf{A}\ \mathbf{where}\ \{\mathsf{O}_0 : \mathsf{A} \to \overline{\varphi_0} \to \gamma_0, \dots_m\})[\mathsf{B}/\mathsf{A}]$ *and* $\mathbf{codata}\ \mathsf{B}\ \mathbf{where}\ \{\mathsf{O}_0 : \mathsf{B} \to \overline{\rho_0} \to \tau_0, \dots_n\}$ *where $m \geq n$ and* $(\overline{\rho_0 <: \varphi_0} \wedge \gamma_0 <: \tau_0) \wedge \dots_n$.

Lemma 14 (Inversion of Typing for $\mathcal{H}_{<:}$). *We have a number of observations based on the conclusion of the typing rules.*

- $\Gamma \vdash x : \tau$ *implies* $x : \varphi \in \Gamma$ *and* $\varphi <: \tau$.
- $\Gamma \vdash \mathbf{fix}\ x{:}\tau\ \mathbf{in}\ e : \tau$ *implies* $\Gamma, x : \tau \vdash e : \tau$.
- $\Gamma \vdash \mathsf{K}(e_0, \dots) : \tau$ *implies all of the following*
  - τ *is some data* X
  - $\mathsf{K} : \rho_0 \dots \to \mathsf{X} \in \Gamma$
  - $\Gamma \vdash e_0 : \rho_0, \dots$.
- $\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{K}_0(\overline{x}_0) \to t_0, \dots\} : \tau$ *implies all of the following*

  - $\Gamma \vdash e : \mathsf{X}$
  - $\mathsf{K}_0 : \overline{\rho}_0 \to \mathsf{X} \in \Gamma$
  - $\Gamma, \overline{x_0 : \varphi_0} \vdash t_0 : \gamma$ *and* $\overline{\rho_0 <: \varphi_0}$ *and* $\gamma <: \tau, \ldots$
- $\Gamma \vdash \{\mathsf{O}_0(\overline{x}_0) \to e_0, \ldots\} : \tau$ *implies all of the following*
  - τ *is some codata* B
  - $\mathsf{B} <: \mathsf{A}$
  - $\mathsf{O}_0 : \mathsf{A} \to \overline{\rho}_0 \to \varphi_0 \in \Gamma$ *and* $\Gamma, \overline{x_0 : \psi_0} \vdash e_0 : \gamma_0$ *and* $\overline{\rho_0 <: \psi_0}$ *and* $\gamma_0 <: \varphi_0, \ldots$
- $\Gamma \vdash e.\mathsf{O}(t_0, \ldots) : \tau$ *implies all of the following*
  - $\mathsf{O} : \mathsf{A} \to \rho_0 \cdots \to \tau \in \Gamma$
  - $\Gamma \vdash e : \mathsf{A}$
  - $\Gamma \vdash t_0 : \rho_0, \ldots$
- $\Gamma \vdash t : \tau$ *implies* $\Gamma \vdash t : \varphi$ *and* $\varphi <: \tau$

Lemma 15 ($\mathcal{H}_{<:}$ Canonical Forms). *For $e \in \mathcal{H}_{<:}$ where e is a value,*

- *If $\Gamma \vdash e : \mathsf{A}$, then e has the form $\{\overline{\mathsf{O}_i(\overline{x}_i) \to t_i}\}$.*
- *If $\Gamma \vdash e : \mathsf{X}$, then e has the form $\mathsf{K}_i(\overline{V})$.*

Theorem 16 ($\mathcal{H}_{<:}$ Progress). *If $\vdash t : \tau$, then either t is a value or $t \mapsto t'$.*

Proof. By induction on the derivation $\vdash t : \tau$.
There is one extra case from the $\mathcal{H}$ progress proof.

- ...
- case $\Gamma \vdash t : \tau$,
  By inversion, $\Gamma \vdash t : \varphi$ and $\varphi <: \tau$.
  By the inductive hypothesis, either t is a value or $t \mapsto t'$.

□

Lemma 17 ($\mathcal{H}_{<:}$ Substitution). *If $\Gamma, x : \tau \vdash t : \rho$ and $\Gamma \vdash e : \tau$, then $\Gamma \vdash t[e/x] : \rho$.*

Theorem 18 ($\mathcal{H}_{<:}$ Preservation). *If $\Gamma \vdash t : \tau$, and $t \mapsto t'$, then $\Gamma \vdash t' : \tau$.*

Proof. By induction on the derivation $\Gamma \vdash t : \tau$.

- case $\Gamma, x : \tau \vdash x : \tau$ is immediate, since there is no $t \mapsto t'$.
- case $\Gamma \vdash \textbf{fix } x{:}\tau \textbf{ in } e : \tau$,
  There is one reduction for t, that is $\textbf{fix } x{:}\tau \textbf{ in } e \mapsto e[\textbf{fix } x{:}\tau \textbf{ in } e/x]$.
  By inversion, we know $\Gamma, x : \tau \vdash e : \tau$.
  By Lemma 11, $\Gamma \vdash e[\textbf{fix } x{:}\tau \textbf{ in } e/x] : \tau$.
- case $\Gamma \vdash \{\overline{\mathsf{O}_i(\overline{x}_i) \to e_i}\} : \tau$ is immediate, since there is no $t \mapsto t'$.
- case $\Gamma, \mathsf{O} : \mathsf{A} \to \overline{\rho} \to \tau \vdash e.\mathsf{O}(\overline{t}) : \tau$,
  By inversion on the typing rule, we know $\Gamma \vdash e : \mathsf{A}$ and $\overline{\Gamma \vdash t : \rho}$.
  There are three reductions to consider:
  - case $e.\mathsf{O}(\overline{t}) \mapsto e'.\mathsf{O}(\overline{t})$, we know $\Gamma \vdash e'.\mathsf{O}(\overline{t}) : \tau$ by the inductive hypothesis, subsumption typing rule, and then the observation typing rule.
  - case $V.\mathsf{O}(\overline{V}, t_i, \overline{t}) \mapsto V.\mathsf{O}(\overline{V}, t_i', \overline{t})$, we know $\Gamma \vdash V.\mathsf{O}(\overline{V}, t_i', \overline{t}) : \tau$ by the inductive hypothesis, subsumption typing rule, and then the observation typing rule.
  - case $\{\ldots, \mathsf{O}(\overline{x}) \to u, \ldots\}.\mathsf{O}(\overline{V}) \mapsto u[\overline{V/x}]$ where $\overline{t} = \overline{V}$,
    By inversion on the typing rule for e, *i.e.* $\Gamma \vdash \{\ldots, \mathsf{O}(\overline{x}) \to u, \ldots\} : \mathsf{B}$, we know $\Gamma, \overline{x : \varphi} \vdash u : \gamma$ and $\overline{\rho <: \varphi}$ and $\gamma <: \tau$.
    By the subsumption typing rule, $\overline{\Gamma \vdash V : \varphi}$
    By Lemma 11, $\Gamma \vdash u[\overline{V/x}] : \gamma$.

Types

$$[\![\mathsf{C}]\!] \quad = \quad \mathsf{C}$$

Environment

$$\begin{array}{rcl} [\![\bullet]\!] & = & \bullet \\ [\![\Gamma, x{:}\mathsf{C}]\!] & = & [\![\Gamma]\!], x{:}[\![\mathsf{C}]\!] \end{array}$$

Fig. 9. Translation of Types and Environments

By the subsumption typing rule, $\Gamma \vdash u_i\overline{[V/x]} : \tau$.

- case $\Gamma, \mathsf{K} : \overline{\tau} \cdots \to \mathsf{X} \vdash \mathsf{K}(\overline{e}) : \mathsf{X}$,
  By inversion on the typing rule, we know $\overline{\Gamma \vdash e : \tau}$.
  There is one reduction rule to consider, $\mathsf{K}(\overline{V}, e_i, \overline{e}) \mapsto \mathsf{K}(\overline{V}, e_i', \overline{e})$.
  By the inductive hypothesis and the constructor typing rule, we know $\Gamma \vdash \mathsf{K}(\overline{V}, e_i', \overline{e}) : \mathsf{X}$.
- case $\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ \{\overline{\mathsf{K}_i(\overline{x}_i) \to t_i}\} : \tau$,
  By inversion on the typing rule, we know $\Gamma \vdash e : \mathsf{X}$ and $\mathsf{K}_0 : \overline{\rho}_0 \to \mathsf{X} \in \Gamma$ and $((\Gamma, \overline{x_0 : \varphi_0} \vdash t_0 : \gamma) \wedge \overline{\rho_0 <: \varphi_0} \wedge (\gamma <: \tau)), \ldots$.
  There are two reductions to consider:
  - case $\mathbf{case}\ e\ \mathbf{of}\ \{\overline{\mathsf{K}_i(\overline{x}_i) \to t_i}\} \mapsto \mathbf{case}\ e'\ \mathbf{of}\ \{\overline{\mathsf{K}_i(\overline{x}_i) \to t_i}\}$, we know $\Gamma \vdash \mathbf{case}\ e'\ \mathbf{of}\ \{\overline{\mathsf{K}_i(\overline{x}_i) \to t_i} : \tau$ by the inductive hypothesis and the case-expression typing rule.
  - case $\mathbf{case}\ \mathsf{K}_i(\overline{V})\ \mathbf{of}\ \{\ldots, \mathsf{K}_i(\overline{x}_i) \to t_i, \ldots\} \mapsto t_i\overline{[V_i/x_i]}$,
    By inversion on the typing rule for e, *i.e.* $\Gamma, \mathsf{K}_i : \overline{\rho}_i \to \mathsf{X} \vdash \mathsf{K}_i(\overline{V}) : \mathsf{X}$, we know that $\overline{\Gamma \vdash V_i : \rho_i}$.
    By the subsumption typing rule, $\overline{\Gamma \vdash V_i : \varphi_i}$.
    By Lemma 11, we know $\Gamma \vdash t_i\overline{[V_i/x_i]} : \gamma$.
    By the subsumption typing rule, $\Gamma \vdash t_i\overline{[V_i/x_i]} : \tau$.
- case $\Gamma \vdash t : \tau$,
  By inversion on the subsumption typing rule, we know $\Gamma \vdash t : \varphi$ and $\varphi <: \tau$.
  By the inductive hypothesis, we know $\Gamma \vdash t' : \varphi$ where $t \mapsto t'$.
  By the subsumption typing rule, we know $\Gamma \vdash t' : \tau$.

□

C CORRECTNESS OF COMPILATION

First, we must specify the translation of FJ types and environments to $\mathcal{H}_{<:}$'s (Figure 9) which were omitted in the main text.

Lemma 19 (Subtype preservation). *If* $\mathsf{A} <: \mathsf{B}$ *for* FJ *types* A, B, *then* $[\![\mathsf{A}]\!] <: [\![\mathsf{B}]\!]$ *for the* $\mathcal{H}_{<:}$ *codata types.*

Proof. By induction on the subtype relation for FJ.

- case $\mathsf{A} <: \mathsf{A}$, is immediate as $\mathcal{H}_{<:}$ contains the same reflexive subtype rule.
- case $\mathsf{A} <: \mathsf{C}$ where $\mathsf{A} <: \mathsf{B}$ and $\mathsf{B} <: \mathsf{C}$, holds with induction and $\mathcal{H}_{<:}$'s similar transitivity rule.
- case $\mathsf{A} <: \mathsf{B}$ where there exists the class declaration **class** A **extends** B $\{\ldots\}$,
  Both A and B are translated into codata declarations. Since A extends B, its translation contains a "casts$^{\#}$" observation and all of the fields and methods of B in addition to its own. Therefore, the breadth subtyping rule for codata gives us $[\![A]\!] <: [\![B]\!]$.

□

Theorem 20 (Type preservation). *If* $\Gamma \vdash t : \mathsf{A}$ *where t contains only upcasts, then* $[\![\Gamma]\!] \vdash \mathcal{T}[\![t]\!] : [\![\mathsf{A}]\!]$.

Proof. By induction on the derivation $\Gamma \vdash t : \mathsf{A}$.

- case $\Gamma, x : \tau \vdash x : \tau$ is immediate because translation of types is the identity.
- case $\Gamma \vdash e.f_i : \mathsf{C}_i$,
  By inversion on the field access typing rule, $\Gamma \vdash e : \mathsf{A}$ and fields(A) $= \mathsf{C}_0\ f_0, \dots$.
  By the inductive hypothesis and $\Gamma \vdash e : \mathsf{A}$, we know $[\![\Gamma]\!] \vdash \mathcal{T}[\![e]\!] : [\![\mathsf{A}]\!]$.
  We know by $\mathcal{T}[\![-]\!]$ on the class definition of A, that $\mathsf{f}_i : \mathsf{A} \to C_i \in [\![\Gamma]\!]$.
  By $\mathcal{H}_{<:}$'s observation rule, $[\![\Gamma]\!] \vdash \mathcal{T}[\![e]\!].\mathsf{f}_i : [\![\mathsf{C}_i]\!]$.
- case $\Gamma \vdash e.\mathsf{m}_i(u_0, \dots) : \mathsf{C}$,
  By inversion on the method invocation rule, $\Gamma \vdash e : \mathsf{A}$, mtype$(\mathsf{m}_i, \mathsf{A}) = \mathsf{D}_0 \cdots \to \mathsf{C}$, and $\Gamma \vdash u_0 : \mathsf{E}_0$ where $\mathsf{E}_0 <: \mathsf{D}_0 \dots$.
  By the inductive hypothesis and $\Gamma \vdash e : \mathsf{A}, \Gamma \vdash u_0 : \mathsf{E}_0, \dots$, we know $[\![\Gamma]\!] \vdash \mathcal{T}[\![e]\!] : [\![\mathsf{A}]\!], [\![\Gamma]\!] \vdash \mathcal{T}[\![u_0]\!] : [\![\mathsf{E}_0]\!], \dots$.
  By Lemma 19, $[\![\mathsf{E}_0]\!] <: [\![\mathsf{D}_0]\!], \dots$.
  By $\mathcal{D}_m[\![-]\!]$, a method of type $\mathsf{D}_0 \cdots \to \mathsf{C}$ is translated to an observation of type $\mathsf{A} \to \mathsf{D}_0 \cdots \to \mathsf{C} \in \mathcal{H}_{<:}$.
  By the $\mathcal{H}_{<:}$ observation, we can build the derivation $[\![\Gamma]\!] \vdash \mathcal{T}[\![e]\!].\mathsf{m}_i([\![u]\!]_0, \dots) : [\![\mathsf{C}]\!]$ after applying the subsumption typing rule to each of the arguments $[\![\mathsf{E}_i]\!] \Rightarrow [\![\mathsf{D}_i]\!]$.
- case $\Gamma \vdash$ **new** $\mathsf{C}(u_0, \dots) : \mathsf{C}$,
  By inversion on the object construction rule, we know fields(C) $= \mathsf{D}_0\ f_0, \dots, \Gamma \vdash u_0 : \mathsf{B}_0$ where $\mathsf{B}_0 <: \mathsf{D}_0, \dots$.
  By the inductive hypothesis, $[\![\Gamma]\!] \vdash \mathcal{T}[\![u_0]\!] : [\![\mathsf{B}_0]\!], \dots$.
  By Lemma 19, $[\![\mathsf{B}_0]\!] <: [\![\mathsf{D}_0]\!], \dots$.
  By $C[\![-]\!]$, an object constructor is turned into a recursive function with the type $\mathsf{D}_0 \cdots \to \mathsf{C} \in \mathcal{H}_{<:}$.
  By $\mathcal{T}[\![-]\!]$, object constructor application is tuned into function application.
  Since functions are macros for observations, we can use the observation typing rule to construct the derivation $[\![\Gamma]\!] \vdash \mathcal{K}[\![\text{constr(C)}]\!]\ \mathcal{T}[\![u_0]\!]\ \cdots : [\![\mathsf{C}]\!]$ after applying the subsumption rule to the arguments $[\![\mathsf{B}_i]\!] \Rightarrow [\![\mathsf{D}_i]\!]$.
- case $\Gamma \vdash (\mathsf{C})\ e : \mathsf{C}$,
  There are three typing rules for casts, but we only need to focus the ones with upcasts.
  By inversion on the typing rule for an upcast, we know $\Gamma \vdash e : \mathsf{D}$.
  By the inductive hypothesis, $[\![\Gamma]\!] \vdash \mathcal{T}[\![e]\!] : [\![\mathsf{D}]\!]$.
  By $\mathcal{T}[\![-]\!]$, casting becomes an if-expression (a macro for case-expressions). The interrogated term is looking up a data type in a list, which is valid forall codata constructed by translating class definitions because they all contain the casts$^{\#}$ observation. The branches of the if statement are $\mathcal{T}[\![e]\!]$ and Ω.
  $[\![\Gamma]\!] \vdash \mathcal{T}[\![e]\!] : [\![\mathsf{D}]\!]$ holds by induction and $[\![\Gamma]\!] \vdash \Omega : [\![\mathsf{D}]\!]$ holds by the definition of the Ω macro.
  Since, (C) e contains only upcasts. The only typing rule that applies is one where $\mathsf{D} <: \mathsf{C}$.
  By Lemma 19, $[\![\mathsf{D}]\!] <: [\![\mathsf{C}]\!]$.
  By the subsumption typing rule, $[\![\Gamma]\!] \vdash \mathcal{T}[\![e]\!] : [\![\mathsf{C}]\!]$.

□

Theorem 21. *If* $t \mapsto^* V$ *for some* $t, V \in$ FJ, *then* $\mathcal{T}[\![t]\!] \mapsto^* W$ *where* W *is a value in* $\mathcal{H}_{<:}$ *such that* $\mathcal{T}[\![V]\!] \mapsto W$.

Proof. This holds by the following observations:

- For any V in FJ, $\mathcal{T}[\![V]\!] \mapsto^* W$, where W is a value in $\mathcal{H}_{<:}$.
  There is only on value in FJ: **new** $\mathsf{C}(\overline{V})$. This is translated into a function application $(\mathcal{K}[\![\text{constr(A)}]\!]\ \overline{\mathcal{T}[\![V]\!]})$ in $\mathcal{H}_{<:}$. This *always* length $\overline{V}$ plus 1 steps to result in a codata type value in $\mathcal{H}_{<:}$.
- For every evaluation context E in FJ, there exists some evaluation context E' in $\mathcal{H}_{<:}$ such that $\mathcal{T}[\![E[t]]\!] = E'[\mathcal{T}[\![t]\!]]$.
  - case $\square$, is trivial as $E' = \square$.
  - case $E.f$, then $E' = E.\mathsf{f}$.
  - case $E.\mathsf{m}(\overline{e})$, then $E' = E.\mathsf{m}_i(\overline{e})$.
  - case $V.\mathsf{m}(V_0, \dots, V_n, E, t_0, \dots, t_m)$, $E' = V.\mathsf{m}(V_0, \dots, V_n, E, t_0, \dots, t_m)$.
  - case **new** $\mathsf{A}(V_0, \dots, V_n, E, t_0, \dots, t_m)$, E' is $\mathcal{K}[\![\text{constr(A)}]\!]\ V_0\ \dots\ V_n\ E\ t_0\ \dots\ t_m$, which matches the observation argument context.
  - case (A) E, E' is **let** $x = E$ **in** $\dots$, which matches the observation argument context, since let-expressions desugar into function application.
- For any $t \mapsto t'$ in FJ, $\mathcal{T}[\![t]\!] \mapsto^* \mathcal{T}[\![t']\!]$.
  - case (**new** $\mathsf{C}(\overline{V})).f_i \mapsto V_i$,
    $\mathcal{T}[\![($**new** $\mathsf{C}(\overline{V})).f_i]\!]$ is $(\mathcal{K}[\![\text{constr(A)}]\!]\ \overline{\mathcal{T}[\![V]\!]}).\mathsf{f}_i$
    By the definition of $\mathcal{K}[\![-]\!]$ and $\mathcal{H}_{<:}$'s dynamic semantics, we know that the second step taken must be the fixed point substitution.
    Next, we take length of $\overline{V}$ steps of copattern matching to get to a codata introduction (this is taking the fields as arguments).
    Finally, one more reduction gets the ith field from the codata type, which is $\mathcal{T}[\![V_i]\!]$.
  - case (**new** $\mathsf{C}(\overline{V})).\mathsf{m}_i(\overline{W}) \mapsto u_i\overline{[W/x]}[$**new** $\mathsf{C}(\overline{V})/$**this**$]$,
    This case is almost identical to the field access case. The difference being that copattern matching returns a term $\mathcal{T}[\![u_i]\!]\overline{[\mathcal{T}[\![W]\!]/x]}$. The substitution for **this** in $\mathcal{T}[\![u_i]\!]$ is handled during the translation $\mathcal{K}[\![-]\!]$.
  - case (A) (**new** $\mathsf{C}(\overline{V})) \mapsto$ **new** $\mathsf{C}(\overline{V})$ where C <: A,
    As with the other two cases, the first steps are to build a codata type by fixed-point substitution and accepting the fields $\overline{V}$ as arguments. Next we take steps evaluating the interrogated term of the if expression by the case-expression evaluation context. If C <: A, then the branch returning the evaluated $\mathcal{T}[\![$**new** $\mathsf{C}(\overline{V})]\!]$ is returned.

□