Codata in Action

Paul Downen[1], Zachary Sullivan[1], Zena M. Ariola[1], and Simon Peyton Jones[2]

[1] University of Oregon, Eugene, USA**
`pdownen@cs.uoregon.edu`, `zsulliva@cs.uoregon.edu`, `ariola@cs.uoregon.edu`
[2] Microsoft Research, Cambridge, UK
`simonpj@microsoft.com`

Abstract. Computer scientists are well-versed in dealing with data structures. The same cannot be said about their dual: codata. Even though codata is pervasive in category theory, universal algebra, and logic, the use of codata for programming has been mainly relegated to representing infinite objects and processes. Our goal is to demonstrate the benefits of codata as a general-purpose programming abstraction independent of any specific language: eager or lazy, statically or dynamically typed, and functional or object-oriented. While codata is not featured in many programming languages today, we show how codata can be easily adopted and implemented by offering simple inter-compilation techniques between data and codata. We believe codata is a common ground between the functional and object-oriented paradigms; ultimately, we hope to utilize the Curry-Howard isomorphism to further bridge the gap.

Keywords: codata · lambda-calculi · encodings · Curry-Howard · function programming · object-oriented programming

1 Introduction

Functional programming enjoys a beautiful connection to logic, known as the Curry-Howard correspondence, or proofs as programs principle [22]; results and notions about a language are translated to those about proofs, and vice-versa [17]. In addition to expressing computation as proof transformations, this connection is also fruitful for education: everybody would understand that the assumption "an x is zero" does not mean "every x is zero," which in turn explains the subtle typing rules for polymorphism in programs. The typing rules for modules are even more cryptic, but knowing that they correspond exactly to the rules for existential quantification certainly gives us more confidence that they are correct! While not everything useful must have a Curry-Howard correspondence, we believe finding these delightful coincidences where the same idea is rediscovered many times in both logic and programming can only be beneficial [43].

One such instance involves *codata*. In contrast with the mystique it has as a programming construct, codata is pervasive in mathematics and logic, where

** This work is supported by the National Science Foundation under grant CCF-1719158.

it arises through the lens of duality. The most visual way to view the duality is in the categorical diagrams of sums versus products—the defining arrows go *into* a sum and come *out of* a product—and in algebras versus coalgebras [25]. In proof theory, codata has had an impact on theorem proving [5] and on the foundation of computation via *polarity* [47, 30]. Polarity recognizes which of two dialogic actors speaks first: the proponent (who seeks to verify or prove a fact) or the opponent (who seeks to refute the fact).

The two-sided, interactive view appears all over the study of programming languages, where data is concerned about how values are constructed and codata is concerned about how they are used [15]. Sometimes, this perspective is readily apparent, like with session types [7] which distinguish internal choice (a provider's decision) versus external choice (a client's decision). But other occurrences are more obscure, like in the semantics of PCF (*i.e.* the call-by-name λ-calculus with numbers and general recursion). In PCF, the result of evaluating a program must be of a ground type in order to respect the laws of functions (namely η) [33]. This is not due to differences between ground types versus "higher types," but to the fact that data types are *directly observable*, whereas codata types are only *indirectly observable* via their interface.

Clearly codata has merit in theoretical pursuits; we think it has merit in practical ones as well. The main application of codata so far has been for representing infinite objects and coinductive proofs in proof assistants [1, 40]. However, we believe that codata also makes for an important general-purpose programming feature. Codata is a bridge between the functional and object-oriented paradigms; a common denominator between the two very different approaches to programming. On one hand, functional languages are typically rich in data types—as many as the programmer wants to define via `data` declarations—but has a paucity of codata types (usually just function types). On the other hand, object-oriented languages are rich in codata types—programmer-defined in terms of classes or interfaces—but a paucity of data types (usually just primitives like booleans and numbers). We illustrate this point with a collection of example applications that arise in both styles of programming, including common encodings, demand-driven programming, abstraction, and Hoare-style reasoning.

While codata types can be seen in the shadows behind many examples of programming—often hand-compiled away by the programmer—not many functional languages have native support for them. To this end, we demonstrate a pair of simple compilation techniques between a typical core functional language (with data types) and one with codata. One direction—based on the well-known visitor pattern from object-oriented programming—simultaneously shows how to extend an object-oriented language with data types (as is done by Scala) and how to compile core functional programs to a more object-oriented setting (*e.g.* targeting a backend like JavaScript or the JVM). The other shows how to add native codata types to functional languages by reducing them to commonly-supported data types and how to compile a "pure" object-oriented style of programming to a functional setting. Both of these techniques are macro-expansions that are not specific to any particular language, as they work with both statically and dynamically typed disciplines, and they preserve the well-typed status of programs without increasing the complexity of the types involved.

Our claim is that codata is a universal programming feature that has been thus-far missing or diminished in today's functional programming languages. This is too bad, since codata is not just a feature invented for the convenience of programmers, but a persistent idea that has sprung up over and over from the study of mathematics, logic, and computation. We aim to demystify codata, and en route, bridge the wide gulf between the functional and object-oriented paradigms. Fortunately, it is easy for most mainstream languages to add or bring out codata today without a radical change to their implementation. But ultimately, we believe that the languages of the future should incorporate *both* data and codata outright. To that end, our contributions are to:

– (Section 2) Illustrate the benefits of codata in both theory and practice: (1) a decomposition of well-known λ-calculus encodings by inverting the priority of construction and destruction; (2) a first-class abstraction mechanism; (3) a method of demand-driven programming; and (4) a static type system for representing Hoare-style invariants on resource use.
– (Section 3) Provide simple transformations for compiling data to codata, and vice-versa, which are appropriate for languages with different evaluation strategies (eager or lazy) and type discipline (static or dynamic).
– (Section 4) Demonstrate various implementations of codata for general-purpose programming in two ways: (1) an extension of Haskell with codata; and (2) a prototype language that compiles to several languages of different evaluation strategies, type disciplines, and paradigms.

2 The Many Faces of Codata

Codata can be used to solve other problems in programming besides representing infinite objects and processes like streams and servers [1, 40]. We start by presenting codata as a merger between theory and practice, whereby *encodings* of data types in an object-oriented style turn 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 a language-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 [16]. Specifying *pre- and post- conditions* of protocols, which is available in some object systems [14], is straightforward with indexed, recursive codata types, *i.e.* objects with guarded methods [41].

2.1 Church Encodings and Object-Oriented Programming

Crucial information structures, like booleans, numbers, and lists 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 [9] 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 `Bool`ean value. However, it turns out that there is a *most-general* eliminator of `Bool`eans: 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 then 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.* [1] 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"`) is `"yes"`, whereas (`false.If "yes" "no"`) is `"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`. 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.

This style of inverting the definition of data types—either into specific codata types or into polymorphic functions—is also related to another concept in object-oriented programming. First, consider how a functional programmer would represent a binary `Tree` (with integer-labeled leaves) and a `walk` function that traverses a tree by converting the labels on all leaves and combining the results of sub-trees:

```
data Tree where Leaf   : Int → Tree
                Branch : Tree → Tree → Tree

walk : (Int → a) → (a → a → a) → Tree → a
walk b f (Leaf x)     = b x
walk b f (Branch l r) = f (walk b f l) (walk b f r)
```

The above code relies on pattern-matching on values of the `Tree` data type and higher-order functions `b` and `f` for accumulating the result. Now, how might an object-oriented programmer tackle the problem of traversing a tree-like structure? The *visitor pattern*! With this pattern, the programmer specifies a “visitor” object which contains knowledge of what to do at every node of the tree, and tree objects must be able to accept a visitor with a method that will recursively walk down each subcomponent of the tree. In a pure style—which returns an accumulated result directly instead of using mutable state as a side channel for results—the visitor pattern for a simple binary tree interface will look like:

```
codata TreeVisitor a where
  VisitLeaf   : TreeVisitor a → (Int → a)
  VisitBranch : TreeVisitor a → (a → a → a)

codata Tree where
  Walk : Tree → (forall a. TreeVisitor a → a)

leaf       : Int → Tree
leaf   x   = {Walk v → v.VisitLeaf x}

branch     : Tree → Tree → Tree
branch l r = {Walk v → v.VisitBranch (l.Walk v) (r.Walk v)}
```

And again, we can write this same code more elegantly, without the need to break apart the two arguments across the equal sign with a manual abstraction, using copatterns as:

```
(leaf     x).Walk v = v.VisitLeaf x
(branch l r).Walk v = v.VisitBranch (l.Walk v) (r.Walk v)
```

Notice how the above code is just an object-oriented presentation of the following encoding of binary trees into the polymorphic λ-calculus:

$$\mathit{Tree} = \forall a. \mathit{TreeVisitor}\ a \to a \qquad \mathit{TreeVisitor}\ a = (\mathit{Int} \to a) \times (a \to a \to a)$$

$$\mathit{leaf} : \mathit{Int} \to \mathit{Tree}$$
$$\mathit{leaf}\ (x{:}\mathit{Int}) = \Lambda a. \lambda v{:}\mathit{TreeVisitor}\ a.\ (\mathit{fst}\ v)\ x$$

$$\mathit{branch} : \forall a. \mathit{Tree} \to \mathit{Tree} \to \mathit{Tree}$$
$$\mathit{branch}\ (l{:}\mathit{Tree})\ (r{:}\mathit{Tree}) = \Lambda a. \lambda v{:}\mathit{TreeVisitor}\ a.\ (\mathit{snd}\ v)\ (l\ a\ v)\ (r\ a\ v)$$

The only essential difference between this λ-encoding of trees versus the λ-encoding of booleans above is currying: the representation of the data type *Tree* takes a single product *TreeVisitor a* of the necessary arguments, whereas the data type *Bool* takes the two necessary arguments separately. Besides this easily-converted difference of currying, the usual Böhm-Berarducci encodings shown here correspond to a pure version of the visitor pattern.

2.2 Demand-Driven Programming

In "Why functional programming matters" [23], 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 [34].

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 an agnostic way which 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.[3]

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 implementor (to change hidden details without disrupting any clients) as well as the client (to ignore details not exposed by the interface).

Reynolds [36] identified, and Cook [12] 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. *What is the language-agnostic representation of procedural abstraction? 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 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.

[3] To see the full code for all the examples of [24] implemented in terms of codata, visit https://github.com/zachsully/codata_examples.

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 [12] 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 operation `fold : (a → b → b) → b → List a → b` is the standard functional fold):

```
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 [44]. 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 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.* [41] 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 [14], where they present an extension to C$^\sharp$ 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`.

Intermezzo 1 This declaration puts one aspect in the hands of the programmer, though. A client can open a socket and never close it, hogging the resource. We can remedy this problem with linear types, which force us to address any loose ends before finishing the program. With linear types, it would be a type error to have a lingering `Live` socket laying around at the end of the program, and a call to `Close` would use it up. Furthermore, linear types would ensure that outdated copies of `Socket` objects cannot be used again, which is especially appropriate for actions like `Bind` which is meant to *transform* a `Raw` socket into a `Bound` one, and likewise for `Connect` which transforms a `Bound` socket into a `Live` one. Even better, enhancing linear types with a more sophisticated notion of ownership—like in the Rust programming language which differentiates a *permanent* transfer of ownership from *temporarily* borrowing it—makes this resource-sensitive interface especially pleasant. Observations like `Bind`, `Connect`, and `Close` which are meant to fully consume the observed object would involve full ownership of the object itself to the method call and effectively replace the old object with the returned one. In contrast, observations like `Send` and `Receive` which are meant to be repeated on the same object would merely borrow the object for the duration of the action so that it could be used again.

3 Inter-compilation of Core Calculi

We saw previously examples of using codata types to replicate well-known encodings of data types into the λ-calculus. Now, let's dive in and show how data and codata types formally relate to one another. In order to demonstrate the relationship, we will consider two small languages that extend the common polymorphic λ-calculus: λ^{data} extends λ with user-defined algebraic data types, and

λ^{codata} extends λ with user-defined codata types. In the end, we will find that both of these foundational languages can be inter-compiled into one another. Data can be represented by codata via the visitor pattern ($\mathfrak{V}$). Codata can be represented by data by tabulating the possible answers of objects ($\mathfrak{T}$).

$$\begin{tikzcd}
\lambda^{data} \arrow[r, bend left, "\text{Visitor } (\mathfrak{V})"] & \lambda^{codata} \arrow[l, bend left, "\text{Tabulate } (\mathfrak{T})"]
\end{tikzcd}$$

In essence, this demonstrates how to compile programs between the functional and object-oriented paradigms. The $\mathfrak{T}$ direction shows how to extend existing functional languages (like OCaml, Haskell, or Racket) with codata objects without changing their underlying representation. Dually, the $\mathfrak{V}$ direction shows how to compile functional programs with data types into an object-oriented target language (like JavaScript).

Each of the encodings are macro expansions, in the sense that they leave the underlying base λ-calculus constructs of functions, applications, and variables unchanged (as opposed to, for example, continuation-passing style translations). They are defined to operate on untyped terms, but they also preserve typability when given well-typed terms. The naïve encodings preserve the operational semantics of the original term, according to a call-by-name semantics. We also illustrate how the encodings can be modified slightly to correctly simulate the call-by-value operational semantics of the source program. To conclude, we show how the languages and encodings can be generalized to more expressive type systems, which include features like existential types and indexed types (*a.k.a.* generalized algebraic data types and guarded methods).

Notation We use both an overline $\overline{t}$ and dots $t_1 \dots$ to indicate a *sequence* of terms t (and likewise for types, variables, *etc.*). The arrow type $\overline{\tau} \to \mathsf{T}$ means $\tau_1 \to \cdots \to \tau_n \to \mathsf{T}$; when n is 0, it is not a function type, *i.e.* just the codomain T. The application $\mathsf{K}\ \overline{t}$ means $(((\mathsf{K}\ t_1)\ \dots)\ t_n)$; when n is 0, it is not a function application, but the constant K. We write a single step of an operational semantics with the arrow $\mapsto$, and many steps (*i.e.* its reflexive-transitive closure) as $\mapsto\!\!\!\!\rightarrow$. Operational steps may occur within an evaluation context E, *i.e.* $t \mapsto t'$ implies that $E[t] \mapsto E[t']$.

3.1 Syntax and Semantics

We present the syntax and semantics of the base language and the two extensions λ^{data} and λ^{codata}. For the sake of simplicity, we keep the languages as minimal as possible to illustrate the main inter-compilations. Therefore, λ^{data} and λ^{codata} do not contain recursion, nested (co)patterns, or indexed types. The extension with recursion is standard, and an explanation of compiling (co)patterns can be found in [40, 39, 11]. Indexed types are later discussed informally in Section 3.6.

Syntax:

$$\begin{aligned}\text{Type} \ni \quad \tau, \rho &::= a \mid \tau \to \rho \mid \forall a.\, \tau \\ \text{Term} \ni t, u, e &::= x \mid t\ u \mid \lambda x.\, e\end{aligned}$$

Operational Semantics:

Call-by-name	Call-by-value
$V ::= x \mid \lambda x.\, e \qquad E ::= \Box \mid E\ u$	$V ::= x \mid \lambda x.\, e \qquad E ::= \Box \mid E\ u \mid V\ E$
$(\lambda x.\, e)\ u \mapsto e[u/x]$	$(\lambda x.\, e)\ V \mapsto e[V/x]$

Type System (where $S = t$ for call-by-name and $S = V$ for call-by-value):

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash t : \tau \to \rho \quad \Gamma \vdash u : \tau}{\Gamma \vdash t\ u : \rho} \qquad \frac{\Gamma, x : \tau \vdash e : \rho}{\Gamma \vdash \lambda x.\, e : \tau \to \rho}$$

$$\frac{\Gamma, a \vdash S : \tau}{\Gamma \vdash S : \forall a.\, \tau} \qquad \frac{\Gamma \vdash t : \forall a.\tau \quad \Gamma \vdash \rho}{\Gamma \vdash t : \tau[\rho/a]}$$

Fig. 1: Polymorphic λ-calculus: The base language

The Base Language We will base both our core languages of interest on a common starting point: the polymorphic λ-calculus as shown in Figure 1.[4] This is the standard simply typed λ-calculus extended with impredicative polymorphism (*a.k.a.* generics). There are only three forms of terms (variables x, applications $t\ u$, and function abstractions $\lambda x.e$) and three forms of types (type variables a, function types $\tau \to \rho$, and polymorphic types $\forall a.\tau$). We keep the type abstraction and instantiation implicit in programs—as opposed to explicit as in System F—for two reasons. First, this more accurately resembles the functional languages in which types are inferred, as opposed to mandatory annotations explicit within the syntax of programs. Second, it more clearly shows how the translations that follow do not rely on first knowing the type of terms, but apply to any untyped term. In other words, the compilation techniques are also appropriate for dynamically typed languages like Scheme and Racket.

Figure 1 reviews both the standard call-by-name and call-by-value operational semantics for the λ-calculus. As usual, the difference between the two is that in call-by-value, the argument of a function call is evaluated prior to substitution, whereas in call-by-name the argument is substituted first. This is implied by the different set of evaluation contexts (E) and the fact that the operational rule uses a more restricted notion of value (V) for substitutable arguments in call-by-value. Note that, there is an interplay between evaluation and typing. In a more general setting where effects are allowed, the typing rule for introducing polymorphism (*i.e.* the rule with $S : \forall a.\tau$ in the conclusion) is only safe for substitutable terms, which imposes the well-known the *value restriction* for call-by-value (limiting S to values), but requires no such restriction in call-by-name where every term is a substitutable value (letting S be any term).

[4] The judgement $\Gamma \vdash \rho$ should be read as: all free type variables in ρ occur in Γ. As usual Γ, a means that a does not occur free in Γ.

Syntax:

$$
\begin{array}{lrl}
\text{Declaration} & \ni \quad d & ::= \textbf{data}\ \mathsf{T}\ \overline{a}\ \textbf{where}\ \mathsf{K} : \overline{\tau} \to \mathsf{T}\ \overline{a}\ \dots \\
\text{Type} & \ni \quad \tau, \rho & ::= a \mid \tau \to \rho \mid \forall a.\, \tau \mid \mathsf{T}\ \overline{\rho} \\
\text{Term} & \ni t, u, e & ::= x \mid t\ u \mid \lambda x.\, e \mid \mathsf{K}\ \overline{t} \mid \textbf{case}\ t\ \{\overline{\mathsf{K}\ \overline{x} \to t}\}
\end{array}
$$

Operational Semantics:

Call-by-name

$$
\begin{array}{l}
V ::= \cdots \mid \mathsf{K}\ \overline{t} \\
E ::= \cdots \mid \textbf{case}\ E\ \{\overline{\mathsf{K}\ \overline{x} \to e}\}
\end{array}
$$

$$\textbf{case}\ (\mathsf{K}\ \overline{t})\ \{\mathsf{K}\ \overline{x} \to e,\ \dots\} \mapsto e\overline{[t/x]}$$

Call-by-value

$$
\begin{array}{l}
V ::= \cdots \mid \mathsf{K}\ \overline{V} \\
E ::= \cdots \mid \textbf{case}\ E\ \{\overline{\mathsf{K}\ \overline{x} \to e}\} \mid \mathsf{K}\ \overline{V}\ E\ \overline{t}
\end{array}
$$

$$\textbf{case}\ (\mathsf{K}\ \overline{V})\ \{\mathsf{K}\ \overline{x} \to e,\ \dots\} \mapsto e\overline{[V/x]}$$

Type System:

$$\frac{\mathsf{K} : \forall \overline{a}.\, \tau_1 \to \cdots \to \mathsf{T}\ \overline{a} \in \Gamma \quad \Gamma \vdash t_1 : \tau_1[\overline{\rho}/\overline{a}] \quad \dots}{\Gamma \vdash \mathsf{K}\ t_1 \cdots : \mathsf{T}\ \overline{\rho}}$$

$$\frac{\Gamma \vdash t : \mathsf{T}\ \overline{\rho} \quad \mathsf{K}_1 : \forall \overline{a}.\, \overline{\tau_1} \to \mathsf{T}\ \overline{a} \in \Gamma \quad \Gamma, \overline{x_1 : \tau_1[\overline{\rho}/\overline{a}]} \vdash e_1 : \tau' \quad \dots}{\Gamma \vdash \textbf{case}\ t\ \{\mathsf{K}_1\ \overline{x_1} \to e_1,\ \dots\} : \tau'}$$

Fig. 2: λ^{data}: Extending polymorphic λ-calculus with data types

A Language with Data The first extension of the λ-calculus is with user-defined data types, as shown in Figure 2; it corresponds to a standard core language for statically typed functional languages. Data declarations introduce a new type constructor (T) as well as some number of associated constructors (K) that build values of that data type. For simplicity, the list of branches in a case expression are considered unordered and non-overlapping (*i.e.* no two branches for the same constructor within a single case expression). The types of constructors are given alongside free variables in Γ, and the typing rule for constructors requires they be fully applied. We also assume an additional side condition to the typing rule for case expressions that the branches are exhaustive (*i.e.* every constructor of the data type in question is covered as a premise).

Figure 2 presents the extension to the operational semantics from Figure 1, which is also standard. The new evaluation rule for data types reduces a case expression matched with an applied constructor. Note that since the branches are unordered, the one matching the constructor is chosen out of the possibilities and the parameters of the constructor are substituted in the branch's pattern. There is also an additional form of constructed values: in call-by-name any constructor application is a value, whereas in call-by-value only constructors parameterized by other values is a value. As such, call-by-value goes on to evaluate constructor parameters in advance, as shown by the extra evaluation context. In both evaluation strategies, there is a new form of evaluation context that points out the discriminant of a case expression, since it is mandatory to determine which constructor was used before deciding the appropriate branch to take.

Syntax:

$$\begin{array}{lll} \text{Declaration} \ni & d ::= \textbf{codata}\ \mathsf{U}\ \overline{a}\ \textbf{where}\ \mathsf{H} : \mathsf{U}\ \overline{a} \to \tau \ldots \\ \text{Type} & \ni \quad \tau, \rho ::= a \mid \tau \to \rho \mid \forall a.\, \tau \mid \mathsf{U}\ \overline{\rho} \\ \text{Term} & \ni t, u, e ::= x \mid t\ u \mid \lambda x.\, e \mid t.\mathsf{H} \mid \{\overline{\mathsf{H} \to e}\} \end{array}$$

Operational Semantics:

Call-by-name

$$V ::= \cdots \mid \{\overline{\mathsf{H} \to e}\} \quad E ::= \cdots \mid E.\mathsf{H}$$

$$\{\mathsf{H} \to e, \ldots\}.\mathsf{H} \mapsto e$$

Call-by-value

$$V ::= \cdots \mid \{\overline{\mathsf{H} \to e}\} \quad E ::= \cdots \mid E.\mathsf{H}$$

$$\{\mathsf{H} \to e, \ldots\}.\mathsf{H} \mapsto e$$

Type System:

$$\frac{\mathsf{H} : \forall \overline{a}.\, \mathsf{U}\ \overline{a} \to \tau \in \Gamma \quad \Gamma \vdash t : \mathsf{U}\ \overline{\rho}}{\Gamma \vdash t.\mathsf{H} : \tau[\overline{\rho}/\overline{a}]} \qquad \frac{\Gamma \vdash \mathsf{H}_1 : \mathsf{U}\ \overline{\rho} \to \tau_1 \quad \Gamma \vdash e_1 : \tau_1 \quad \ldots}{\Gamma \vdash \{\mathsf{H}_1 \to e_1, \ldots\} : \mathsf{U}\ \overline{\rho}}$$

Fig. 3: λ^{codata}: Extending polymorphic λ-calculus with codata types

A Language with Codata The second extension of the λ-calculus is with user-defined codata types, as shown in Figure 3. Codata declarations in λ^{codata} define a new type constructor (U) along with some number of associated destructors (H) for projecting responses out of values of a codata type. The type level of λ^{codata} corresponds directly to λ^{data}. However, at the term level, we have codata observations of the form $t.\mathsf{H}$ using "dot notation", which can be thought of as sending the message H to the object t or as a method invocation from object-oriented languages. Values of codata types are introduced in the form $\{\mathsf{H}_1 \to e_1, \ldots, \mathsf{H}_n \to e_n\}$, which lists each response this value gives to all the possible destructors of the type. As with case expressions, we take the branches to be unordered and non-overlapping for simplicity.

Interestingly, the extension of the operational semantics with codata—the values, evaluation contexts, and reduction rules—are identical for both call-by-name and call-by-value evaluation. In either evaluation strategy, a codata object $\{\mathsf{H} \to e, \ldots\}$ is considered a value and the codata observation $t.\mathsf{H}$ *must* evaluate t no matter what to continue, leading to the same form of evaluation context $E.\mathsf{H}$. The additional evaluation rule selects and invokes the matching branch of a codata object and is the same regardless of the evaluation strategy.

Note that the reason that values of codata types are the same in any evaluation strategy is due to the fact that the branches of the object are only ever evaluated on-demand, *i.e.* when they are observed by a destructor, similar to the fact that the body of a function is only ever evaluated when the function is called. This is the semantic difference that separates codata types from records found in many programming languages. Records typically map a collection of labels to a collection of values, which are evaluated in advance in a call-by-value language similar to the constructed values of data types. Whereas with codata objects, labels map to *behavior* which is only invoked when observed.

$$\mathfrak{V}\left[\!\!\left[\begin{array}{l}\textbf{data}\ \mathsf{T}\ \overline{a}\ \textbf{where}\\ \quad \mathsf{K}_1 : \overline{\tau_1} \to \mathsf{T}\ \overline{a}\\ \quad\quad \vdots\\ \quad \mathsf{K}_n : \overline{\tau_n} \to \mathsf{T}\ \overline{a}\end{array}\right]\!\!\right] = \begin{array}{l}\textbf{codata}\ \mathsf{T}_{visit}\ \overline{a}\ b\ \textbf{where}\\ \quad \mathsf{K}_1 : \mathsf{T}_{visit}\ \overline{a}\ b \to \overline{\tau_1} \to b\\ \quad\quad \vdots\\ \quad \mathsf{K}_n : \mathsf{T}_{visit}\ \overline{a}\ b \to \overline{\tau_n} \to b\\ \textbf{codata}\ \mathsf{T}\ \overline{a}\ \textbf{where}\\ \quad \mathsf{Case}_{\mathsf{T}} : \mathsf{T}\ \overline{a} \to \forall b.\, \mathsf{T}_{visit}\ \overline{a}\ b \to b\end{array}$$

$$\mathfrak{V}[\![\mathsf{K}_i\ \overline{t}]\!] = \{\mathsf{Case}_{\mathsf{T}} \to \lambda v.\, (v.\mathsf{K}_i)\ \overline{\mathfrak{V}[\![t]\!]}\}$$

$$\mathfrak{V}[\![\textbf{case}\ t\ \{\mathsf{K}_1\ \overline{x_1} \to e_1, \ldots\}]\!] = (\mathfrak{V}[\![t]\!].\mathsf{Case}_{\mathsf{T}})\ \{\mathsf{K}_1 \to \lambda\overline{x_1}.\, \mathfrak{V}[\![e_1]\!], \ldots\}$$

Fig. 4: $\mathfrak{V} : \lambda^{data} \to \lambda^{codata}$ mapping data to codata via the visitor pattern

The additional typing rules for λ^{codata} are also given in Figure 3. The rule for typing $t.\mathsf{H}$ is analogous to a combination of type instantiation and application, when viewing H as a function of the given type. The rule for typing a codata object, in contrast, is similar to the rule for typing a case expression of a data type. However, in this comparison, the rule for objects is partially "upside down" in the sense that the primary type in question ($\mathsf{U}\ \overline{\rho}$) appears in the conclusion rather than as a premise. This is the reason why there is one less premise for typing codata objects than there is for typing data case expressions. As with that rule, we assume that the branches are exhaustive, so that every destructor of the codata type appears in the premise.

3.2 Compiling Data to Codata: The Visitor Pattern

In Section 2.1, we illustrated how to convert a data type representing trees into a codata type. This encoding corresponds to a rephrasing of the object-oriented visitor pattern to avoid unnecessary side-effects. Now lets look more generally at the pattern, to see how any algebraic data type in λ^{data} can be encoded in terms of codata in λ^{codata}.

The visitor pattern has the net effect of inverting the orientation of a data declaration (wherein construction comes first) into codata declarations (wherein destruction comes first). This reorientation can be used for compiling user-defined data types in λ^{data} to codata types in λ^{codata} as shown in Figure 4. As with all of the translations we will consider, this is a macro expansion since the syntactic forms from the base λ-calculus are treated homomorphically (*i.e.* $\mathfrak{V}[\![\lambda x.\, e]\!] = \lambda x.\, \mathfrak{V}[\![e]\!]$, $\mathfrak{V}[\![t\ u]\!] = \mathfrak{V}[\![t]\!]\ \mathfrak{V}[\![u]\!]$, and $\mathfrak{V}[\![x]\!] = x$). Furthermore, this translation also perfectly preserves types, since the types of terms are exactly the same after translation (*i.e.* $\mathfrak{V}[\![\tau]\!] = \tau$).

Notice how each data type ($\mathsf{T}\ \overline{a}$) gets represented by *two* codata types: the "visitor" ($\mathsf{T}_{visit}\ \overline{a}\ b$) which says what to do with values made with each constructor, and the type itself ($\mathsf{T}\ \overline{a}$) which has one method which accepts a visitor and returns a value of type b. An object of the codata type, then, must be capable of accepting *any* visitor, no matter what type of result it returns. Also notice that we include no other methods in the codata type representation of $\mathsf{T}\ \overline{a}$.

At the level of terms, first consider how the case expression of the data type is encoded. The branches of the case (contained within the curly braces) are represented as a first-class object of the visitor type: each constructor is mapped to the corresponding destructor of the same name and the variables bound in the pattern are mapped to parameters of the function returned by the object in each case. The whole case expression itself is then implemented by calling the sole method ($\mathsf{Case}_{\mathsf{T}}$) of the codata object and passing the branches of the case as the corresponding visitor object. Shifting focus to the constructors, we can now see that they are compiled as objects that invoke the corresponding destructor on any given visitor, and the terms which were parameters to the constructor are now parameters to a given visitor's destructor. Of course, other uses of the visitor pattern might involve a codata type (T) with more methods implementing additional functionality besides case analysis. However, we only need the one method to represent data types in λ^{data} because case expressions are *the* primitive destructor for values of data types in the language.

For example, consider applying the above visitor pattern to a binary tree data type as follows:

$$\mathfrak{V}\left[\!\!\left[\begin{array}{l}\textbf{data}\ \mathsf{Tree}\ \textbf{where}\\ \quad \mathsf{Leaf}\quad\ : \mathsf{Int} \to \mathsf{Tree}\\ \quad \mathsf{Branch} : \mathsf{Tree} \to \mathsf{Tree} \to \mathsf{Tree}\end{array}\right]\!\!\right] = \begin{array}{l}\textbf{codata}\ \mathsf{Tree}_{visit}\ b\ \textbf{where}\\ \quad \mathsf{Leaf}\quad\ : \mathsf{Int} \to b\\ \quad \mathsf{Branch} : \mathsf{Tree} \to \mathsf{Tree} \to b\\ \textbf{codata}\ \mathsf{Tree}\ \textbf{where}\\ \quad \mathsf{Case}_{\mathsf{Tree}} : \mathsf{Tree} \to \forall b.\, \mathsf{Tree}_{visit}\ b \to b\end{array}$$

$$\mathfrak{V}[\![\mathsf{Leaf}\ n]\!] = \{\mathsf{Case}_{\mathsf{Tree}} \to \lambda v.\, v.\mathsf{Leaf}\ n\}$$

$$\mathfrak{V}[\![\mathsf{Branch}\ l\ r]\!] = \{\mathsf{Case}_{\mathsf{Tree}} \to \lambda v.\, v.\mathsf{Branch}\ l\ r\}$$

$$\mathfrak{V}\left[\!\!\left[\textbf{case}\ t\ \left\{\begin{array}{l}\mathsf{Leaf}\ n \quad\ \to e_l\\ \mathsf{Branch}\ l\ r \to e_b\end{array}\right\}\right]\!\!\right] = \mathfrak{V}[\![t]\!].\mathsf{Case}_{\mathsf{Tree}}\ \left\{\begin{array}{l}\mathsf{Leaf}\quad\ \to \lambda n.\, \mathfrak{V}[\![e_l]\!]\\ \mathsf{Branch} \to \lambda l.\, \lambda r.\, \mathfrak{V}[\![e_b]\!]\end{array}\right\}$$

Note how this encoding differs from the one that was given in Section 2.1 since the $\mathsf{Case}_{\mathsf{Tree}}$ method is non-recursive whereas the $\mathsf{Walk}_{\mathsf{Tree}}$ method was recursive, in order to model a depth-first search traversal of the tree.

Of course, other operations, like the `walk` function, could be written in terms of case expressions and recursion as usual by an encoding with above method calls. However, it is possible to go one step further and include other primitive destructors—like recursors or iterators in the style of Gödel's system T—by embedding them as other methods of the encoded codata type. For example, we can represent `walk` as a primitive destructor as it was in Section 2.1 *in addition* to non-recursive case analysis by adding an alternative visitor Tree_{walk} and one more destructor to the generated Tree codata type like so:

$$\begin{array}{l}\textbf{codata}\ \mathsf{Tree}_{walk}\ b\ \textbf{where}\\ \quad \mathsf{Leaf}\quad\ : \mathsf{Int} \to b\\ \quad \mathsf{Branch} : b \to b \to b\end{array} \qquad \begin{array}{l}\textbf{codata}\ \mathsf{Tree}\ \textbf{where}\\ \quad \mathsf{Case}_{\mathsf{Tree}}\ : \mathsf{Tree} \to \forall b.\, \mathsf{Tree}_{visit}\ b \to b\\ \quad \mathsf{Walk}_{\mathsf{Tree}} : \mathsf{Tree} \to \forall b.\, \mathsf{Tree}_{walk}\ b \to b\end{array}$$

$$\mathfrak{V}[\![\mathsf{Leaf}\ n]\!] = \left\{\begin{array}{l}\mathsf{Case}_{\mathsf{Tree}} \to \lambda v.\, v.\mathsf{Leaf}\ n\\ \mathsf{Walk}_{\mathsf{Tree}} \to \lambda w.\, w.\mathsf{Leaf}\ n\end{array}\right\}$$

$$\mathfrak{V}[\![\mathsf{Branch}\ l\ r]\!] = \left\{\begin{array}{l}\mathsf{Case}_{\mathsf{Tree}} \to \lambda v.\, v.\mathsf{Branch}\ l\ r\\ \mathsf{Walk}_{\mathsf{Tree}} \to \lambda w.\, w.\mathsf{Branch}\ (l.\mathsf{Walk}_{\mathsf{Tree}})\ (r.\mathsf{Walk}_{\mathsf{Tree}})\end{array}\right\}$$

For codata types with n destructors, where $n \geq 1$:

$$\mathfrak{T}\left[\!\!\left[\begin{array}{c}\textbf{codata}\ \mathsf{U}\ \overline{a}\ \textbf{where}\\ \mathsf{H}_1 : \mathsf{U}\ \overline{a} \to \tau_1\\ \vdots\\ \mathsf{H}_n : \mathsf{U}\ \overline{a} \to \tau_n\end{array}\right]\!\!\right] = \begin{array}{l}\textbf{data}\ \mathsf{U}\ \overline{a}\ \textbf{where}\\ \quad \mathsf{Table}_{\mathsf{U}} : \tau_1 \to \cdots \to \tau_n \to \mathsf{U}\ \overline{a}\end{array}$$

$$\mathfrak{T}[\![t.\mathsf{H}_i]\!] = \textbf{case}\ \mathfrak{T}[\![t]\!]\ \{\mathsf{Table}_{\mathsf{U}}\ y_1 \dots y_n \to y_i\}$$

$$\mathfrak{T}[\![\{\mathsf{H}_1 \to e_1, \dots, \mathsf{H}_n \to e_n\}]\!] = \mathsf{Table}_{\mathsf{U}}\ \mathfrak{T}[\![e_1]\!] \dots \mathfrak{T}[\![e_n]\!]$$

For codata types with 0 destructors (where Unit is the same for every such U):

$$\mathfrak{T}\left[\!\!\left[\begin{array}{c}\textbf{codata}\ \mathsf{U}\ \overline{a}\ \textbf{where}\\ \text{--}\textit{no destructors}\end{array}\right]\!\!\right] = \begin{array}{l}\textbf{data}\ \mathsf{Unit}\ \textbf{where}\\ \quad \mathsf{unit} : \mathsf{Unit}\end{array}$$

$$\mathfrak{T}[\![\{\}]\!] = \mathsf{unit}$$

Fig. 5: $\mathfrak{T} : \lambda^{codata} \to \lambda^{data}$ tabulating codata responses with data tuples

where the definition of Tree_{visit} and the encoding of case expressions is the same. In other words, this compilation technique can generalize to as many primitive observations and recursion schemes as desired.

3.3 Compiling Codata to Data: Tabulation

Having seen how to compile data to codata, how can we go the other way? The reverse compilation would be useful for extending functional languages with user-defined codata types, since many functional languages are compiled to a core representation based on the λ-calculus with data types.

Intuitively, the declared data types in λ^{data} can be thought of as "sums of products." In contrast, the declared codata types in λ^{codata} can be thought of as "products of functions." Since both core languages are based on the λ-calculus, which has higher-order functions, the main challenge is to relate the two notions of "products." The codata sense of products are based on projections out of abstract objects, where the different parts are viewed individually and only when demanded. The data sense of products, instead, are based on tuples, in which all components are laid out in advance in a single concrete structure.

One way to convert codata to data is to *tabulate* an object's potential answers ahead of time into a data structure. This is analogous to the fact that a function of type `Bool` → `String` can be alternatively represented by a tuple of type `String * String`, where the first and second components are the responses of the original function to `true` and `false`, respectively. This idea can be applied to λ^{codata} in general as shown in the compilation in Figure 5.

A codata declaration of U becomes a data declaration with a single constructor ($\mathsf{Table}_{\mathsf{U}}$) representing a tuple containing the response for each of the original destructors of U. At the term level, a codata abstraction is compiled by concretely tabulating each of its responses into a tuple using the $\mathsf{Table}_{\mathsf{U}}$ constructor. A destructor application returns the specific component of the constructed tuple which corresponds to that projection. Note that, since we assume that each object is exhaustive, the tabulation transformation is relatively straightforward; filling in "missing" method definitions with some error value that can be stored in the tuple at the appropriate index would be done in advance as a separate pre-processing step.

Also notice that there is a special case for non-observable "empty" codata types, which are all collapsed into a single pre-designated Unit data type. The reason for this collapse is to ensure that this compilation preserves typability: if applied to a well-typed term, the result is also well-typed. The complication arises from the fact that when faced with an empty object $\{\}$, we have no idea which constructor to use without being given further typing information. So rather than force type checking or annotation in advance for this one degenerate case, we instead collapse them all into a single data type so that there is no need to differentiate based on the type. In contrast, the translation of non-empty objects is straightforward, since we can use the name of any one of the destructors to determine the codata type it is associated with, which then informs us of the correct constructor to use.

3.4 Correctness

For the inter-compilations between λ^{codata} into λ^{data} to be useful in practice, they should preserve the semantics of programs. For now, we focus only on the call-by-name semantics for each of the languages. With the static aspect of the semantics, this means they should preserve the typing of terms.

Proposition 1 (Type Preservation). *For each of the $\mathfrak{V}$ and $\mathfrak{T}$ translations: if $\Gamma \vdash t : \tau$ then $[\![\Gamma]\!] \vdash [\![t]\!] : [\![\tau]\!]$ (in the call-by-name type system).*

Proof (Sketch). By induction on the typing derivation of $\Gamma \vdash t : \tau$.

With the dynamic aspect of the semantics, the translations should preserve the outcome of evaluation (either converging to some value, diverging into an infinite loop, or getting stuck) for both typed and untyped terms. This works because each translation preserves the reduction steps, values, and evaluation contexts of the source calculus' call-by-name operational semantics.

Proposition 2 (Evaluation Preservation). *For each of the $\mathfrak{V}$ and $\mathfrak{T}$ translations: $t \mapsto\!\!\!\!\rightarrow V$ if and only if $[\![t]\!] \mapsto\!\!\!\!\rightarrow [\![V]\!]$ (in the call-by-name semantics).*

Proof (Sketch). The forward ("only if") implication is a result of the following facts that hold for each translation in the call-by-name semantics:

- For any redex t in the source, if $t \mapsto t'$ then $[\![t]\!] \mapsto t'' \mapsto\!\!\!\!\rightarrow [\![t']\!]$.
- For any value V in the source, $[\![V]\!]$ is a value.
- For any evaluation context E in the source, there is an evaluation context E' in the target such that $[\![E[t]]\!] = E'[[\![t]\!]]$ for all t.

The reverse ("if") implication then follows from the fact that the call-by-name operational semantics of both source and target languages is deterministic.

3.5 Call-by-Value: Correcting the Evaluation Order

The presented inter-compilation techniques are correct for the call-by-name semantics of the calculi. But what about the call-by-value semantics? It turns out that the simple translations seen so far do not correctly preserve the call-by-value semantics of programs, but they can be easily fixed by being more careful about how they treat the values of the source and target calculi. In other words, we need to make sure that values are translated to values, and evaluation contexts to evaluation contexts. For instance, the following translation (up to renaming) does not preserve the call-by-value semantics of the source program:

$$\mathfrak{T}[\![\{\mathsf{Fst} \to \mathit{error}, \mathsf{Snd} \to \mathsf{True}\}]\!] = \mathsf{Pair}\ \mathit{error}\ \mathsf{True}$$

The object $\{\mathsf{Fst} \to \mathit{error}, \mathsf{Snd} \to \mathsf{True}\}$ is a value in call-by-value, and the erroneous response to the Fst will only be evaluated when observed. However, the structure $\mathsf{Pair}\ \mathit{error}\ \mathsf{True}$ is not a value in call-by-value, because the field error must be evaluated in advance which causes an error immediately. In the other direction, we could also have

$$\mathfrak{V}[\![\mathsf{Pair}\ \mathit{error}\ \mathsf{True}]\!] = \{\mathsf{Case} \to \lambda v.\, v.\mathsf{Pair}\ \mathit{error}\ \mathsf{True}\}$$

Here, the immediate error in $\mathsf{Pair}\ \mathit{error}\ \mathsf{True}$ has become incorrectly delayed inside the value $\{\mathsf{Case} \to \lambda v.\, v.\mathsf{Pair}\ \mathit{error}\ \mathsf{True}\}$.

The solution to this problem is straightforward: we must manually delay computations that are lifted out of (object or λ) abstractions, and manually force computations before their results are hidden underneath abstractions. For the visitor pattern, the correction is to only introduce the codata object on constructed values. We can handle other constructed terms by naming their non-value components in the style of administrative-normalization like so:

$$\begin{aligned}
\mathfrak{V}[\![\mathsf{K}_i\ \overline{V}]\!] &= \{\mathsf{Case}_\mathsf{T} \to \lambda v.\, v.\mathsf{K}_i\ \overline{V}\} \\
\mathfrak{V}[\![\mathsf{K}_i\ \overline{V}\ u\ \overline{t}]\!] &= \mathbf{let}\ x = u\ \mathbf{in}\ \mathfrak{V}[\![\mathsf{K}_i\ \overline{V}\ x\ \overline{t}]\!] \qquad \text{if } u \text{ is not a value}
\end{aligned}$$

Conversely, the tabulating translation $\mathfrak{T}$ will cause the on-demand observations of the object to be converted to preemptive components of a tuple structure. To counter this change in evaluation order, a thunking technique can be employed as follows:

$$\begin{aligned}
\mathfrak{T}[\![t.\mathsf{H}_i]\!] &= \mathbf{case}\ \mathfrak{T}[\![t]\!]\ \{\mathsf{Table}_\mathsf{U}\ y_1 \ldots y_n \to \mathbf{force}\, y_i\} \\
\mathfrak{T}[\![\{\mathsf{H}_1 \to e_1, \ldots, \mathsf{H}_n \to e_n\}]\!] &= \mathsf{Table}_\mathsf{U}\ (\mathbf{delay}\, \mathfrak{T}[\![e_1]\!]) \ldots (\mathbf{delay}\, \mathfrak{T}[\![e_n]\!])
\end{aligned}$$

The two operations can be implemented as $\mathbf{delay}\, t = \lambda z.\, t$ and $\mathbf{force}\, t = t\ \mathsf{unit}$ as usual, but can also be implemented as more efficient memoizing operations. With all these corrections, Propositions 1 and 2 also hold for the call-by-value type system and operational semantics.

3.6 Indexed Data and Codata Types: Type Equalities

In the world of types, we have so far only formally addressed inter-compilation between languages with simple and polymorphic types. What about the compilation of indexed data and codata types? It turns out some of the compilation techniques we have discussed so far extend to type indexes without further effort, whereas others need some extra help. In particular, the visitor-pattern-based translation $\mathfrak{V}$ can just be applied straightforwardly to indexed data types:

$$\mathfrak{V}\left[\!\!\left[\begin{array}{l}\textbf{data}\ \mathsf{T}\ \overline{a}\ \textbf{where} \\ \quad \mathsf{K}_1 : \overline{\tau_1} \to \mathsf{T}\ \overline{\rho_1} \\ \qquad\quad \vdots \\ \quad \mathsf{K}_n : \overline{\tau_n} \to \mathsf{T}\ \overline{\rho_n}\end{array}\right]\!\!\right] = \begin{array}{l}\textbf{codata}\ \mathsf{T}_{visit}\ \overline{a}\ b\ \textbf{where} \\ \quad \mathsf{K}_1 : \mathsf{T}_{visit}\ \overline{\rho_1}\ b \to \overline{\tau_1} \to b \\ \qquad\qquad \vdots \\ \quad \mathsf{K}_n : \mathsf{T}_{visit}\ \overline{\rho_n}\ b \to \overline{\tau_n} \to b \\ \textbf{codata}\ \mathsf{T}\ \overline{a}\ \textbf{where} \\ \quad \mathsf{Case}_{\mathsf{T}} : \mathsf{T}\ \overline{a} \to \forall b.\, \mathsf{T}_{visit}\ \overline{a}\ b \to b\end{array}$$

In this case, the notion of an indexed visitor codata type exactly corresponds to the mechanics of case expressions for GADTs. In contrast, the tabulation translation $\mathfrak{T}$ does not correctly capture the semantics of indexed codata types, if applied naïvely.

Thankfully, there is a straightforward way of "simplifying" indexed data types to more conventional data types using some built-in support for *type equalities*. The idea is that a constructor with a more specific return type can be replaced with a conventional constructor that is parameterized by type equalities that *prove* that the normal return type must be the more specific one. The same idea can be applied to indexed codata types as well. A destructor that can only act on a more specific instance of the codata type can instead be replaced by one which works on any instance, but then immediately asks for *proof* that the object's type is the more specific one before completing the observation. These two translations, of replacing type indexes with type equalities, are defined as:

$$\mathfrak{Eq}\left[\!\!\left[\begin{array}{l}\textbf{data}\ \mathsf{T}\ \overline{a}\ \textbf{where} \\ \quad \mathsf{K}_1 : \overline{\tau_1} \to \mathsf{T}\ \overline{\rho_1} \\ \qquad\quad \vdots \\ \quad \mathsf{K}_n : \overline{\tau_n} \to \mathsf{T}\ \overline{\rho_n}\end{array}\right]\!\!\right] = \begin{array}{l}\textbf{data}\ \mathsf{T}\ \overline{a}\ \textbf{where} \\ \quad \mathsf{K}_1 : \overline{a \equiv \rho_1} \to \overline{\tau_1} \to \mathsf{T}\ \overline{a} \\ \qquad\qquad \vdots \\ \quad \mathsf{K}_n : \overline{a \equiv \rho_n} \to \overline{\tau_n} \to \mathsf{T}\ \overline{a}\end{array}$$

$$\mathfrak{Eq}\left[\!\!\left[\begin{array}{l}\textbf{codata}\ \mathsf{U}\ \overline{a}\ \textbf{where} \\ \quad \mathsf{H}_1 : \mathsf{U}\ \overline{\rho_1} \to \overline{\tau_1} \\ \qquad\quad \vdots \\ \quad \mathsf{H}_n : \mathsf{U}\ \overline{\rho_n} \to \overline{\tau_n}\end{array}\right]\!\!\right] = \begin{array}{l}\textbf{codata}\ \mathsf{U}\ \overline{a}\ \textbf{where} \\ \quad \mathsf{H}_1 : \mathsf{U}\ \overline{a} \to \overline{a \equiv \rho_1} \to \overline{\tau_1} \\ \qquad\qquad \vdots \\ \quad \mathsf{H}_n : \mathsf{U}\ \overline{a} \to \overline{a \equiv \rho_n} \to \overline{\tau_n}\end{array}$$

This formalizes the intuition that indexed data types can be thought of as *enriching* constructors to carry around additional constraints that were available at their time of construction, whereas indexed codata types can be thought of as *guarding* methods with additional constraints that must be satisfied before an observation can be made. Two of the most basic examples of this simplification are for the type declarations which capture the notion of type equality as an indexed data or indexed codata type, which are defined and simplified like so:

$$\mathfrak{Eq}\left[\!\!\left[\begin{array}{l}\textbf{data}\ \mathsf{Eq}\ a\ b\ \textbf{where} \\ \quad \mathsf{Refl} : \mathsf{Eq}\ a\ a\end{array}\right]\!\!\right] = \begin{array}{l}\textbf{data}\ \mathsf{Eq}\ a\ b\ \textbf{where} \\ \quad \mathsf{Refl} : a \equiv b \to \mathsf{Eq}\ a\ b\end{array}$$

$$\mathfrak{Eq}\left[\!\!\left[\begin{array}{l}\textbf{codata}\ \mathsf{IfEq}\ a\ b\ c\ \textbf{where}\\ \quad \mathsf{AssumeEq} : \mathsf{IfEq}\ a\ a\ c \to c\end{array}\right]\!\!\right] = \begin{array}{l}\textbf{codata}\ \mathsf{IfEq}\ a\ b\ c\ \textbf{where}\\ \quad \mathsf{AssumeEq} : \mathsf{IfEq}\ a\ b\ c \to a \equiv b \to c\end{array}$$

With the above ability to simplify away type indexes, *all* of the presented compilation techniques are easily generalized to indexed data and codata types by composing them with $\mathfrak{Eq}$. For practical programming example, consider the following safe stack codata type indexed by its number of elements.

$$\begin{array}{l}\textbf{codata}\ \mathsf{Stack}\ a\ \textbf{where}\\ \quad \mathsf{Pop}\ \ : \mathsf{Stack}\ (\mathsf{Succ}\ a) \to (\mathbb{Z}, \mathsf{Stack}\ a)\\ \quad \mathsf{Push} : \mathsf{Stack}\ a \to \mathbb{Z} \to \mathsf{Stack}\ (\mathsf{Succ}\ a)\end{array}$$

This stack type is safe in the sense that the Pop operation can only be applied to non-empty Stacks. We cannot compile this to a data type via $\mathfrak{T}$ directly, because that translation does not apply to indexed codata types. However, if we first simplify the Stack type via $\mathfrak{Eq}$, we learn that we can replace the type of the Pop destructor with $\mathsf{Pop} : \mathsf{Stack}\ a \to \forall b.a \equiv \mathsf{Succ}\ b \to (\mathbb{Z}, \mathsf{Stack}\ b)$, whereas the Push destructor is already simple, so it can be left alone. That way, for any object $s : \mathsf{Stack\ Zero}$, even though a client can initiate the observation $s.\mathsf{Pop}$, it will never be completed since there is no way to choose a b and prove that Zero equals Succ b. Therefore, the net result of the combined $\mathfrak{T} \circ \mathfrak{Eq}$ translation turns Stack into the following data type, after some further simplification:

$$\begin{array}{l}\textbf{data}\ \mathsf{Stack}\ a\ \textbf{where}\\ \quad \mathsf{MkS} : (\forall b.a \equiv \mathsf{Succ}\ b \to (\mathbb{Z}, \mathsf{Stack}\ b)) \to (\mathbb{Z} \to \mathsf{Stack}\ (\mathsf{Succ}\ a)) \to \mathsf{Stack}\ a\end{array}$$

Notice how the constructor of this type has two fields; one for Pop and one for Push, respectively. However, the Pop operation is guarded by a proof obligation: the client can only receive the top integer and remaining stack if he/she proves that the original stack contains a non-zero number of elements.

4 Compilation in Practice

We have shown how data and codata are related through the use of two different core calculi. To explore how these ideas manifest in practice, we have implemented codata in a couple of settings. First, we extended Haskell with codata in order to compare the lazy and codata approaches to demand-driven programming described in Section 2.2.[5] Second, we have created a prototype language with indexed (co)data types to further explore the interaction between the compilation and target languages. The prototype language does not commit to a particular evaluation strategy, typing discipline, or paradigm; instead this decision is made when compiling a program to one of several backends. The supported backends include functional ones—Haskell (call-by-need, static types), OCaml (call-by-value, static types), and Racket (call-by-value, dynamic types)—as well as the object-oriented JavaScript.[6] The following issues of complex copattern matching and sharing applies to both implementations; the performance results on efficiency of memoized codata objects are tested with the Haskell extension for the comparison with conventional Haskell code.

[5] The GHC fork is at https://github.com/zachsully/ghc/tree/codata-macro.
[6] The prototype compiler is at https://github.com/zachsully/dl/tree/esop2019.

n	Time(s) codata	Time(s) data	Allocs(bytes) codata	Allocs(bytes) data
10000	0.02	0.01	10,143,608	6,877,048
100000	0.39	0.27	495,593,464	463,025,832
1000000	19.64	18.54	44,430,524,144	44,104,487,488

Table 1: Fibonacci scaling tests for the GHC implementation

Complex Copattern Matching Our implementations support nested copatterns so that objects can respond to chains of multiple observations, even though λ^{codata} only provides flat copatterns. This extension does not enhance the language expressivity but allows more succinct programs [2]. A flattening step is needed to compile nested copatterns down to a core calculus, which has been explored in previous work by Setzer *et al.* [38] and Thibodeau [40] and implemented in OCaml by Regis-Gianas and Laforgue [34]. Their flattening algorithm requires copatterns to completely cover the object's possible observations because the coverage information is used to drive flattening. This approach was refined and incorporated in a dependently typed setting by Cockx and Abel [11]. With our goal of supporting codata independently of typing discipline and coverage analysis, we have implemented the purely syntax driven approach to flattening found in [39]. For example, the `prune` function from Section 2.2 expands to:

```
prune = λx → λt →
  { Node     → t.Node,
    Children → case x of
                 0 → []
                 _ → map (prune(x-1)) t.Children }
```

Sharing If codata is to be used instead of laziness for demand-driven programming, then it must have the same performance characteristics, which relies on sharing the results of computations [6]. To test this, we compare the performance of calculating streams of Fibonacci numbers—the poster child for sharing—implemented with both lazy list data types and a stream codata type in Haskell extended with codata. These tests, presented in Table 1, show the speed of the codata version is always slower in terms of run time and allocations than the lazy list version, but the difference is small and the two versions scale at the same rate. These performance tests are evidence that codata shares the same information when compiled to a call-by-need language; this we get for free because call-by-need data constructors—which codata is compiled into via $\mathfrak{T}$—memoize their fields. In an eager setting, it is enough to use memoized versions of **delay** and **force**, which are introduced by the call-by-value compilation described in Section 3.5. This sharing is confirmed by the OCaml and Racket backends of the prototype language which find the 100th Fibonacci in less than a second (a task that takes hours without sharing).

As the object-oriented representative, the JavaScript backend is a compilation from data to codata using the visitor pattern presented in Section 3.2. Because codata remains codata (*i.e.* JavaScript objects), an optimization must be performed to ensure the same amount of sharing of codata as the other back-

Syntax

$$\text{Values} \ni \quad V ::= \cdots \mid \{\overline{\mathsf{H} \to V}\}$$
$$\text{Terms} \ni t, u, e ::= \cdots \mid t.\mathsf{H} \mid \{\overline{\mathsf{H} \to V}\} \mid \mathbf{let}_{\text{need}}\ \overline{x = t}\ \mathbf{in}\ e$$

Transformation

$$\mathcal{A}[\![t.\mathsf{H}]\!] = \mathcal{A}[\![t]\!].\mathsf{H}$$
$$\mathcal{A}[\![\{\overline{\mathsf{H} \to t}\}]\!] = \mathbf{let}_{\text{need}}\ \overline{x = \mathcal{A}[\![t]\!]}\ \mathbf{in}\ \{\overline{\mathsf{H} \to x}\}$$

Fig. 6: Memoization of λ^{codata}

ends. The solution is to lift out the branches of a codata object, as shown in Figure 6, where the call-by-need let-bindings can be implemented by **delay** and **force** in strict languages as usual. It turns out that this transformation is also needed in an alternative compilation technique presented by Regis-Gianas and Laforgue [34] where codata is compiled to functions, *i.e.* another form of codata.

5 Related Work

Our work follows in spirit of Amin *et al.*'s [3] desire to provide a minimal theory that can model type parameterization, modules, objects and classes. Another approach to combine type parameterization and modules is also offered by 1ML [37], which is mapped to System F. Amin *et al.*'s work goes one step further by translating System F to a calculus that directly supports objects and classes. Our approach differs in methodology: instead of searching for a logical foundation of a pre-determined notion of objects, we let the logic guide us while exploring what objects are. Even though there is no unanimous consensus that functional and object-oriented paradigms should be combined, there have been several hybrid languages for combining both styles of programming, including Scala, the Common Lisp Object System [8], Objective ML [35], and a proposed but unimplemented object system for Haskell [31].

Arising out of the correspondence between programming languages, category theory, and universal algebras, Hagino [20] first proposed codata as an extension to ML to remedy the asymmetry created by data types. In the same way that data types represent initial F-algebras, codata types represent final F-coalgebras. These structures were implemented in the categorical programming language Charity [10]. On the logical side of the correspondence, codata arises naturally in the sequent calculus [46, 29, 15] since it provides the right setting to talk about construction of either the provider (*i.e.* the term) or the client (*i.e.* the context) side of a computation, and has roots in classical [13, 42] and linear logic [18, 19].

In session-typed languages, which also have a foundation in linear logic, external choice can be seen as a codata (product) type dual to the way internal choice corresponds to a data (sum) type. It is interesting that similar problems arise in both settings. Balzer and Pfenning [7] discuss an issue that shows up in choosing between internal and external choice; this corresponds to choosing between data and codata, known as the *expression problem.* They [7] also suggest using the visitor pattern to remedy having external choice (codata) without internal choice (data) as we do in Section 3.2. Of course, session types go beyond codata by adding a notion of temporality (via linearity) and multiple processes that communicate over channels.

To explore programming with coinductive types, Ancona and Zucca [4] and Jeannin *et al.* [26] extended Java and OCaml with regular cyclic structures; these have a finite representation that can be eagerly evaluated and fully stored in memory. A less restricted method of programming these structures was introduced by Abel *et al.* [1, 2] who popularized the idea of programming by observations, *i.e.* using copatterns. This line of work further developed the functionality of codata types in dependently typed languages by adding indexed codata types [41] and dependent copattern matching [11], which enabled the specification of bisimulation proofs and encodings of productive infinite objects in Agda. We build on these foundations by developing codata in practical languages.

Focusing on implementation, Regis-Gianas and Laforgue [34] added codata with a macro transformation in OCaml. As it turns out, this macro definition corresponds to one of the popular encodings of objects in the λ-calculus [27], where codata/objects are compiled to functions from tagged messages to method bodies. This compilation scheme requires the use of GADTs for static type checking, and is therefore only applicable to dynamically typed languages or the few statically typed languages with expressive enough type systems like Haskell, OCaml, and dependently typed languages. Another popular technique for encoding codata/objects is presented in [32], corresponding to a class-based organization of dynamic dispatch [21], and is presented in this paper. This technique compiles codata/objects to products of methods, which has the advantage of being applicable in a simply-typed setting.

6 Conclusion

We have shown here how codata can be put to use to capture several practical programming idioms and applications, besides just modeling infinite structures. In order to help incorporate codata in today's programming languages, we have shown how to compile between two core languages: one based on the familiar notion of data types from functional languages such as Haskell and OCaml, and the other one, based on the notion of a structure defined by reactions to observations [1]. This paper works toward the goal of providing common ground between the functional and object-oriented paradigms; as future work, we would like to extend the core with other features of full-fledged functional and object-oriented languages. A better understanding of codata clarifies both the theory and practice of programming languages. Indeed, this work is guiding us in the use of fully-extensional functions for the compilation of Haskell programs. The design is motivated by the desire to improve optimizations, in particular the ones relying on the "arity" of functions, to be more compositional and work between higher-order abstractions. It is interesting that the deepening of our understanding of objects is helping us in better compiling functional languages!

References

1. Abel, A., Pientka, B., Thibodeau, D., Setzer, A.: Copatterns: Programming infinite structures by observations. In: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 27–38. POPL '13 (2013)
2. Abel, A.M., Pientka, B.: Wellfounded recursion with copatterns: A unified approach to termination and productivity. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. pp. 185–196. ICFP '13 (2013)
3. Amin, N., Rompf, T., Odersky, M.: Foundations of path-dependent types. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. pp. 233–249 (2014)
4. Ancona, D., Zucca, E.: Corecursive featherweight java. In: Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs, FTfJP 2012, Beijing, China, June 12, 2012. pp. 3–10 (2012)
5. Andreoli, J.M.: Logic programming with focusing proofs in linear logic. Journal of Logic and Computation **2**, 297–347 (1992)
6. Ariola, Z.M., Felleisen, M.: The call-by-need lambda calculus. Journal of Functional Programming **7**(3), 265–301 (1997)
7. Balzer, S., Pfenning, F.: Objects as session-typed processes. In: Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control. pp. 13–24. AGERE! 2015, ACM, New York, NY, USA (2015)
8. Bobrow, D.G., Kahn, K.M., Kiczales, G., Masinter, L., Stefik, M., Zdybel, F.: Commonloops: Merging Lisp and object-oriented programming. In: Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), Portland, Oregon, USA, Proceedings. pp. 17–29 (1986)
9. Böhm, C., Berarducci, A.: Automatic synthesis of typed lambda-programs on term algebras. Theoretical Computer Science **39**, 135–154 (1985)
10. Cockett, R., Fukushima, T.: About Charity. Tech. rep., University of Calgary (1992)
11. Cockx, J., Abel, A.: Elaborating dependent (co)pattern matching. In: Proceedings of the 23rd ACM SIGPLAN International Conference on Functional Programming. pp. 75:1–75:30. ICFP '18 (2018)
12. Cook, W.R.: On understanding data abstraction, revisited. In: Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications. pp. 557–572 (2009)
13. Curien, P.L., Herbelin, H.: The duality of computation. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming. pp. 233–243. ICFP '00, ACM, New York, NY, USA (2000)
14. DeLine, R., Fähndrich, M.: Typestates for objects. In: ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings. pp. 465–490 (2004)
15. Downen, P., Ariola, Z.M.: The duality of construction. In: Programming Languages and Systems: 23rd European Symposium on Programming, ESOP 2014. vol. 8410, pp. 249–269. Springer Berlin Heidelberg (Apr 2014)
16. Dummett, M.: The logical basis of methaphysics. In: The William James Lectures, 1976. Harvard University Press, Cambridge, Massachusetts (1991)
17. Gallier, J.: Constructive logics: Part I: A tutorial on proof systems and typed lambda-calculi. Theoretical Computer Science **110**(2), 249–339 (Mar 1993)

18. Girard, J.Y.: Linear logic. Theoretical Computer Science **50**(1), 1–101 (1987)
19. Girard, J.Y.: On the unity of logic. Annals of Pure and Applied Logic **59**(3), 201–217 (1993)
20. Hagino, T.: Codatatypes in ML. Journal of Symbolic Computation pp. 629–650 (1989)
21. Harper, R.: Practical Foundations for Programming Languages. Cambridge University Press, New York, NY, USA, 2nd edn. (2016)
22. Howard, W.A.: The formulae-as-types notion of construction. In: To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, pp. 479–490. Academic Press (1980), unpublished manuscript of 1969
23. Hughes, J.: Why functional programming matters. Computer Journal **32**(2), 98–107 (1989)
24. Hughes, R.J.M.: Super-combinators: a new implementation method for applicative languages. In: Proc. ACM Symposium on Lisp and Functional Programming. pp. 1–10 (1982)
25. Jacobs, B., Rutten, J.: A tutorial on (co)algebras and (co)induction. EATCS Bulletin **62**, 62–222 (1997)
26. Jeannin, J., Kozen, D., Silva, A.: Cocaml: Functional programming with regular coinductive types. Fundam. Inform. **150**(3-4), 347–377 (2017)
27. Krishnamurthi, S.: Programming Languages: Application and Interpretation (2007)
28. Launchbury, J., Peyton Jones, S.L.: State in Haskell. LISP and Symbolic Computation **8**(4), 293–341 (Dec 1995)
29. Munch-Maccagnoni, G.: Focalisation and classical realisability. In: Grädel, E., Kahle, R. (eds.) Computer Science Logic: 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL. pp. 409–423. CSL 2009, Springer Berlin Heidelberg, Berlin, Heidelberg (Sep 2009)
30. Munch-Maccagnoni, G.: Syntax and Models of a non-Associative Composition of Programs and Proofs. Ph.D. thesis, Université Paris Diderot (2013)
31. Nordlander, J.: Polymorphic subtyping in O'Haskell. Science of Computer Programming **43**(2-3), 93–127 (2002)
32. Pierce, B.C.: Types and Programming Languages. The MIT Press (2002)
33. Plotkin, G.: LCF considered as a programming language. Theoretical Computer Science **5**(3), 223 – 255 (1977)
34. Regis-Gianas, Y., Laforgue, P.: 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)
35. Rémy, D., Vouillon, J.: Objective ML: A simple object-oriented extension of ML. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 40–53. POPL '97, ACM, New York, NY, USA (1997)
36. Reynolds, J.C.: User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction, pp. 309–317. Springer (1978)
37. Rossberg, A.: 1ML – core and modules united (F-ing first-class modules). In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming. pp. 35–47. ICFP 2015, ACM, New York, NY, USA (2015)
38. Setzer, A., Abel, A., Pientka, B., Thibodeau, D.: Unnesting of copatterns. In: Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014. pp. 31–45 (2014)
39. Sullivan, Z.: The Essence of Codata and Its Implementation. Master's thesis, University of Oregon (2018)

40. Thibodeau, D.: Programming Infinite Structures using Copatterns. Master's thesis, McGill University (2015)
41. Thibodeau, D., Cave, A., Pientka, B.: Indexed codata types. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming. pp. 351–363 (2016)
42. Wadler, P.: Call-by-value is dual to call-by-name. In: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming. pp. 189–201 (2003)
43. Wadler, P.: Propositions as types. Communications of the ACM **58**(12), 75–84 (Nov 2015)
44. Weirich, S.: Depending on types. In: Proceedings of the 19rd ACM SIGPLAN International Conference on Functional Programming. ICFP '14 (2014)
45. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 224–235. POPL '03 (2003)
46. Zeilberger, N.: On the unity of duality. Annals of Pure and Applied Logic pp. 66–96 (2008)
47. Zeilberger, N.: The Logical Basis of Evaluation Order and Pattern-Matching. Ph.D. thesis, Carnegie Mellon University (2009)

A Statically Checked Finalization with Regions

Intermezzo 1 discusses how linear types can be used to statically enforce even more of the socket protocol by ensuring that stale socket references are consumed after binding, connecting, and closing, and that every socket is closed by the time it goes out of scope. While linear types are necessary for preventing references to stale socket states, there is an alternative technique besides linearity which ensures that sockets must be closed by the end of a client transaction, after which the socket is no longer usable. The idea is to use an extra layer of indirection similar to Haskell's `ST` monad [28].

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

codata Transaction r where
  Run : Transaction r → forall s. Socket s Raw → SK s (s, r)

runTransaction : Transaction r → r
```

First, each of the `Socket` methods are tagged with an additional region parameter (`s`) and return their result within an abstract monad `SK s` of socket actions which prevents socket actions from being run directly outside of a transaction. Second, a socket `Transaction` starts with a `Raw` socket in some region and yields a result `r` along with the region `s` which signifies that the socket has been closed. Finally, the function `runTransaction` generates a fresh `Raw` socket to run the `Transaction` and the associated `SK` action, discards the final closure token `s`, and returns the result `r`. By having a computation return the correct region (`s`) along with the result, we guarantee that the `Transaction` must have closed the socket before exiting. These kinds of protocol-like applications are a natural fit with indexed codata types. All of the behavior of the protocol itself, and the restricted handling of sensitive resources, is expressed in the types, so that the type system can check that the methods are applied correctly.

B Decomposing λ-encodings

Here, we show formally how a small modification of the visitor pattern gives rise to a decomposition of the Church (or Böhm-Berarducci, when types are taken into account) encodings of data. The key point is that by currying the visitor pattern, we only need to use objects with a single method (denoted by the sub-language $\lambda^{codata1}$ where each codata type can only have one destructor), so that the individual identity of codata types can be easily anonymized as just

$$\mathfrak{C}\left[\!\!\left[\begin{array}{l}\textbf{data}\ \mathsf{T}\ \overline{a}\ \textbf{where}\\ \quad \mathsf{K}_1 : \overline{\tau_1} \to \mathsf{T}\ \overline{a}\\ \quad\quad \vdots \\ \quad \mathsf{K}_n : \overline{\tau_n} \to \mathsf{T}\ \overline{a}\end{array}\right]\!\!\right] = \begin{array}{l}\textbf{codata}\ \mathsf{T}\ \overline{a}\ \textbf{where}\\ \quad \mathsf{Case}_\mathsf{T} : \mathsf{T}\ \overline{a} \to \forall b.\, (\overline{\tau}_1 \to b)\\ \qquad\qquad\qquad \to \ldots\\ \qquad\qquad\qquad \to (\overline{\tau_n} \to b)\\ \qquad\qquad\qquad \to b\end{array}$$

$$\mathfrak{C}[\![\mathsf{K}_i\ \overline{t}]\!] = \{\mathsf{Case}_\mathsf{T} \to \lambda y_1 \ldots \lambda y_n.\, y_i\ \overline{\mathfrak{C}[\![t]\!]}\}$$

$$\mathfrak{C}[\![\textbf{case}\ t\ \{\mathsf{K}_1\ \overline{x_1} \to e_1, \ldots\}]\!] = \mathfrak{C}[\![t]\!].\mathsf{Case}_\mathsf{T}\ (\lambda\overline{x_1}.\, \mathfrak{C}[\![e_1]\!])\ \ldots$$

Fig. 7: $\mathfrak{C} : \lambda^{data} \to \lambda^{codata1}$ mapping data to codata with a single destructor

higher-order functions.

$$\begin{tikzcd}
\lambda^{data} \arrow[rr, "\text{Curried Visitor } (\mathfrak{C})"] \arrow[dr, swap, "\text{Church/B.-B. } (\Lambda)"] & & \lambda^{codata1} \arrow[dl, "\text{Anonymize } (\mathfrak{A})"] \\
& \lambda &
\end{tikzcd}$$

In addition to relating the visitor pattern to data and codata, Section 2.1 also argued that common encodings of data types into the (typed or untyped) λ-calculus is decomposed by first encoding data into codata, and then anonymizing that codata type's identity. How can this idea be formalized in terms of a chain of translations starting from λ^{data}, passing through λ^{codata}, and ending at the pure λ-calculus? The main insight is that because functions are just a specific codata type—one with exactly one destructor corresponding to function application—the anonymization step relies on mapping *all* one-destructor codata types into the single function type.

But the visitor pattern, as defined previously, does not yet fit this restriction! Consider Figure 4 again. Indeed, the data type T is translated to a codata type of just one destructor for performing case analysis. However, that destructor refers to another codata type T_v, with one destructor for every constructor of the original data type, for representing the branches of a case. How can we simplify this translation to avoid multiple destructors? Notice how the offending visitor type T_v is effectively a form of user-defined product, and it only ever appears as the argument to the Case_T destructor. What is the standard way of eliminating a compound product argument? Currying! If we inline the auxiliary T_v type into the codata definition of T by separating each component of T_v as a separate argument to Case_T, we get the alternative form of the visitor pattern translation shown in Figure 7. Besides the use of currying, this is the same as the translation in Section 3.2.

Having mapped data types into the simpler $\lambda^{codata1}$, the rest of the path to the λ-calculus is fairly straightforward. The only thing that remains to be done is to translate each user-defined codata type into a higher-order function as shown in Figure 8. At the type level, this involves inlining the definition of each codata type. At the term level, this involves erasing invocations of codata

$$\mathfrak{A}[\![\mathsf{U}\ \overline{\rho}]\!]_\Gamma = \mathfrak{A}[\![\tau]\!]_\Gamma \quad \text{if } \Gamma \vdash \mathsf{H} : \mathsf{U}\ \overline{\rho} \to \tau \qquad \mathfrak{A}[\![t.\mathsf{H}]\!] = t \quad \mathfrak{A}[\![\{\mathsf{H} \to t\}]\!] = \mathfrak{A}[\![t]\!]$$

Fig. 8: $\mathfrak{A} : \lambda^{codata1} \to \lambda$ anonymizing single-method codata as functions

destructors and single-method codata abstractions. Note that, due to the fact that the definition of each codata type is inlined, this particular translation is only well-defined for non-recursive codata types.

The encoding of data types in λ^{data} inside the λ-calculus can now be defined as the composition of these two previous steps: $\Lambda = \mathfrak{A} \circ \mathfrak{C}$. Expanding the definitions, the Λ encoding is:

$$\begin{aligned}
\Lambda[\![\mathsf{T}\ \overline{\rho}]\!] &= \left(\forall b.\, (\overline{\Lambda[\![\tau_1]\!]} \to b) \to \cdots \to (\overline{\Lambda[\![\tau_n]\!]} \to b) \to b\right) \left[\overline{\Lambda[\![\rho]\!]/a}\right] \\
\Lambda[\![\mathsf{K}_i\ \overline{t}]\!] &= \lambda y_1 \ldots \lambda y_n . y_i\ \overline{\Lambda[\![t]\!]} \\
\Lambda[\![\mathbf{case}\ t\ \{\mathsf{K}_1\ \overline{x_1} \to e_1, \ldots\}]\!] &= \Lambda[\![t]\!]\ (\lambda \overline{x_1}.\, \Lambda[\![e_1]\!])\ \ldots
\end{aligned}$$

Note that this encoding is exactly the Church (for untyped) and Böhm-Berarducci (for typed) encodings for finite types (like sums, products, *etc.*). If we wished to extend the encodings to recursive types, we would have to manually extend anonymization ($\mathfrak{A}$) to cover some chosen notion of well-founded type recursion (like the "strictly positive" restriction for inductive types). For example, encoding the `walk` down a tree as in Section 2.1 and 3.2 corresponds to the Church and Böhm-Berarducci encodings of the strictly positive binary tree data type, which works by replacing recursive occurrences of the main type with the generic return type.

Both the $\mathfrak{C}$ and $\mathfrak{A}$ encodings are correct in terms of Propositions 1 and 2 for the call-by-name semantics, but need some modifications for call-by-value. The curried form of the visitor pattern, $\mathfrak{C}$, requires the same correction as $\mathfrak{V}$ in Section 3.5 correction as well, but it *also* must be careful with constant constructors that take no arguments. For these the branches of case expressions matching those constructors, the $\mathfrak{C}$ translation presented in Figure 7 will not insert any λ-abstractions, thereby exposing the branch to be evaluated anyway even if not ultimately taken. To be careful about this possibility, we can manually delay each branch of a case so that it is only evaluated if needed:

$$\begin{aligned}
\mathfrak{C}[\![\mathsf{K}_i\ \overline{V}]\!] &= \{\mathsf{Case}_\mathsf{T} \to \lambda y_1 \ldots \lambda y_n.\ \mathbf{force}\,(y_i\ \overline{\mathfrak{C}[\![V]\!]})\} \\
\mathfrak{C}[\![\mathbf{case}\ t\ \{\mathsf{K}_1\ \overline{x_1} \to e_1, \ldots\}]\!] &= \mathfrak{C}[\![t]\!].\mathsf{Case}_\mathsf{T}\ (\lambda \overline{x_1}.\ \mathbf{delay}\,\mathfrak{C}[\![e_1]\!])\ \ldots
\end{aligned}$$

Similarly, the anonymizing translation $\mathfrak{A}$ also requires the use of delay and force to convert codata objects to delayed thunks.

$$\mathfrak{A}[\![t.\mathsf{H}]\!] = \mathbf{force}\, t \qquad\qquad \mathfrak{A}[\![\{\mathsf{H} \to t\}]\!] = \mathbf{delay}\,\mathfrak{A}[\![t]\!]$$

$$\mathfrak{F}\left[\!\!\left[\begin{array}{c}\textbf{codata}\ \mathsf{U}\ \overline{a}\ \textbf{where}\\ \mathsf{H}_1 : \mathsf{U}\ \overline{\rho_1} \to \tau_1 \\ \vdots \\ \mathsf{H}_n : \mathsf{U}\ \overline{\rho_n} \to \tau_n\end{array}\right]\!\!\right] = \begin{array}{l}\textbf{data}\ \mathsf{U}_{message}\ \overline{a}\ b\ \textbf{where}\\ \quad \mathsf{H}_1 : \mathsf{U}_{message}\ \overline{\rho_1}\ \tau_1 \\ \qquad\qquad \vdots \\ \quad \mathsf{H}_n : \mathsf{U}_{message}\ \overline{\rho_n}\ \tau_n \\ \textbf{data}\ \mathsf{U}\ \overline{a}\ \textbf{where}\\ \quad \mathsf{Object}_{\mathsf{T}} : (\forall b.\, \mathsf{U}_{message}\ \overline{a}\ b \to b) \to \mathsf{U}\ \overline{a}\end{array}$$

$$\mathfrak{F}[\![t.\mathsf{H}_i]\!] = \textbf{case}\ \mathfrak{F}[\![t]\!]\ \{\mathsf{Object}_{\mathsf{T}}\ o \to o\ \mathsf{H}_i\}$$

$$\mathfrak{F}[\![\{\mathsf{H}_1 \to e_1, \ldots\}]\!] = \mathsf{Object}_{\mathsf{T}}\ (\lambda m.\, \textbf{case}\ m\ \{\mathsf{H}_1 \to \mathfrak{F}[\![e_1]\!], \ldots\})$$

Fig. 9: $\mathfrak{F} : \lambda^{codata} \to \lambda^{data}$ mapping codata to dependent functions.

C Compiling Codata to Data: Dependent functions

Since the semantics of codata form the *operational* core of an object-oriented language, the encodings of codata types into languages with data types are doing the same work as existing encodings of objects. There are two popular encodings of objects: as multi-entry functions that branch based on a message argument and as products of functions where messages pick the correct observation to return. The former is described by Krishnamurthi [27] and latter by Harper [21] and Pierce [32]. While the multi-entry function technique is easy enough to define for dynamically typed languages, finding a way to fit it within a static type discipline is much more challenging, since the return type of the function *depends* on the particular entry-point taken (*i.e.* the particular message given to it).

Regis-Gianas and Laforgue [34] use the multi-entry function encoding of codata as presented in Figure 9. Each destructor of the original U codata type is represented as a constructor of the $\mathsf{U}_{message}$ data type representing messages. Note that the return type of the destructor is recorded along-side the parameters to U, which makes $\mathsf{U}_{message}$ into GADT even if the original U was not. The codata type U is then encoded as a data type with a single constructor containing only a function from messages to the appropriate response type for that particular message.

Because codata is translated into a function that waits for a message symbol, this encoding automatically preserves the on-demand semantics of codata in both call-by-name and call-by-value languages. Being behind a λ, the branches of codata types do not automatically get shared, but the same memoization optimization seen in Figure 6 can guarantee sharing by giving names to the branches.

The largest shortcoming of this approach is that (while it works operationally in many languages) type checking the generated code goes beyond simple types or polymorphism, as noted by Xi *et al.* [45]. One way to type-check the result of the encoding is to use indexed functions—a limited form of dependent function which uses GADT to represent the argument type—as was done by Regis-Gianas and Laforgue [34]. The consequence of using such a powerful language construct is that this encoding is only applicable in languages with a rich enough type system, such as OCaml, Haskell, and those with dependent types.

To alleviate the dependence on rich types, we can eliminate the GADT definition of message symbols of the $\mathsf{U}_{message}$ type by instead recording the response of the function to each message inside a large product. This results in a product (where each component is most likely a function) as described in Section 3.3 that follows Harper's class-based organization of dynamic dispatch [21], giving the same encoding as Pierce [32].