Copattern matching and first-class observations in OCaml, with a macro

Paul Laforgue, Yann Régis-Gianas

Copattern matching and first-class observations in OCaml, with a macro

Paul Laforgue
Univ Paris Diderot, Sorbonne Paris Cité
Paris, France
Northeastern University
Boston, U.S.A

Yann Régis-Gianas
Univ Paris Diderot, Sorbonne Paris Cité, IRIF/PPS, UMR 8243 CNRS, PiR2, INRIA Paris-Rocquencourt
Paris, France

ABSTRACT

Infinite data structures are elegantly defined by means of copattern matching, a dual construction to pattern matching that expresses the outcomes of the observations of an infinite structure. We extend the OCaml programming language with copatterns, exploiting the duality between pattern matching and copattern matching. Provided that a functional programming language has GADTs, every copattern matching can be transformed into a pattern matching via a purely local syntactic transformation, a macro. The development of this extension leads us to a generalization of previous calculus of copatterns: the introduction of first-class observation queries. We study this extension both from a formal and practical point of view.

ACM Reference format:
Paul Laforgue and Yann Régis-Gianas. 2017. Copattern matching and first-class observations in OCaml,
with a macro. In *Proceedings of PPDP'17, Namur, Belgium, October 9–11, 2017,* 12 pages.
DOI: 10.1145/3131851.3131869

1 INTRODUCTION

In a strict language, infinite structures are notoriously tricky to define. Consider the following naive definition of the Fibonacci sequence:

```
let rec fib () = 0 :: 1 :: map2 (+) (fib ()) (tl (fib ()))
```

As soon as *fib* is applied, this computation never returns. On the contrary, a well-behaved infinite computation must give the control back after a finite number of computation steps. Lazy evaluation is a way to achieve this:

```
type 'a lazy_list = C of 'a * 'a lazy_list Lazy.t
let rec fib = C (0, lazy (C (1, lazy (map2' (+) fib (tl' fib)))))
```

Here, the second argument of the *C* constructor for lazy lists is a lazy computation, which, by definition, takes the control only if forced by the evaluation context.

The multiple insertions of the **lazy** markers stray the definition of the Fibonacci sequence from its mathematical counterpart since they introduce low-level considerations about the way this definition must be evaluated. Besides, even if this is not enforced by the type checker, algebraic datatypes are intended to model inductive objects, which these lazy lists are not. This problem witnesses the lack of support for infinite structures in the OCaml[10] programming language.

Recently, the duality between induction and coinduction have been put into work to introduce a *copattern matching*[1] mechanism in functional languages and proof assistants. While a pattern matching destructs a finite value defined using a constructor, a copattern matching creates an infinite computation defined in terms of its answers to *observations* performed by the evaluation context. Which of the term or its evaluation context owns the control is central in that duality: a pattern matching owns the control whereas a copattern matching only owns the definitions of the potential outcomes and lets the evaluation context chooses between them. In other words, observations are to infinite structures what *Lazy.force* is to lazy computation.

To illustrate this style of definition-by-observations, consider the following definition of the Fibonacci sequence:

```
let corec fib : int stream with
  | ..#Head → 0
  | ..#Tail : int stream with
  | ..#Tail#Head → 1
  | ..#Tail#Tail → map2 (+) fib (fib#Tail)
```

On the first line, the keyword "**let corec**" introduces a toplevel value named *fib* which is a stream of integers defined using three rules. On the second line, the first rule says that when the *Head* of *fib* is observed, 0 is returned. The second rule is about two chained observations: when the *Tail* of *fib* is observed and the *Head* of this tail is observed, 1 is returned. Finally, the third rule defines the rest of the stream as a stream of integers made by the point-wise addition of *fib* and the stream resulting from the observation of its tail. As shown by this example, a copattern describes how a value is destructed by the evaluation context and a copattern matching branch maps this destruction to the computation it triggers.

The typechecking of a copattern matching differs from the typechecking of standard pattern matching. Indeed, while all the branches of a pattern matching must have the same type, the branches of a copattern matching may not. As a matter of fact, the type of a branch body depends on the type of its corresponding observation: if the head of a stream of integers is observed, the branch must be of type int but for an observation of its tail, the branch must be of type int stream. This dependency between the observation and the type of its outcome is made explicit by the declaration of the codatatype of stream:

PPDP'17, Namur, Belgium

978-1-4503-5291-8/17/10…$15.00
DOI: 10.1145/3131851.3131869

```
type 'a stream = {
  Head : 'a          ← 'a stream;
  Tail : 'a stream ← 'a stream
}
```

What if we now wanted to implement copatterns in OCaml? Can the duality between pattern matching and copattern matching be exploited to reinterpret copattern matchings in terms of pattern matchings? If such a translation exists, is it type-preserving? Does it generate reasonably efficient code?

In this paper, we answer positively to all these questions. As our title suggests, the extension of OCaml with copatterns is actually as straightforward as a macro-expansion: the translation from OCaml with copatterns to OCaml without them is purely local and syntactic. With no surprise, it crucially exploits the duality between functions defined by pattern matching and functions that define codata by copattern matching, going from the second to the first by introducing a well-chosen (and well-typed!) inversion of control. This duality has already been described in prior work about focusing[11, 17]: to quote Licata et al. [11], "values of negative polarity can be introduced by pattern matching against their destructors". We apply this idea on a mainstream programming language and we generalize the theory to deal with indexed codatatypes.

Two aspects of this translation are a bit more surprising. First, even to extend vanilla ML with non-indexed codatatypes, the typechecking of the compiled copattern matchings requires sophisticated types, namely Generalized Algebraic Datatypes[16] and second-order polymorphic types[6]. Second, our translation reveals that observations can actually be reified as first-class first-order objects in the source language . In this paper, we study this translation both from a formal and practical point of view.

We introduce a typed core language named λ^{C} that captures the syntax and semantics of ML with GADTs extended with indexed codatatypes, copatterns and reified observations. The language presentation emphasizes the duality exploited by the translation. We present a local type preserving translation from λ^{C} to λ^{G}, a core language with GADTs and second-order polymorphic values in the style of Rémy's and Garrigue's type system[6] (Section 3).

This translation gives rise to a lightweight extension of the OCaml language. We implemented a proof-of-concept compiler which produces efficient code thanks to the insertion of appropriate **lazy** markers in the generated code (Section 4).

Finally, we present a comonadic implementation of the Game of Life using observation-based programming in order to illustrate our extension of OCaml with copatterns and to exhibit what first-class observations can be good for (Section 5).

2 OVERVIEW

The translation we informally present in this section is based on a simple idea: to turn a copattern matching into a standard pattern matching, it suffices to transfer, at the observation site, the control from the evaluation context to the observed codata. To let the (translated) codata decide which computation must be triggered, the evaluation context must communicate the intented observation as a value that can be analyzed by the (translated) codata using a pattern matching.

As a consequence, the translation of a codatatype declaration must include a datatype declaration for *observation queries* in addition to the datatype declaration for the translated codatatype itself. As an illustration, the type declaration for observation queries on streams is:

```
type ('o, 'a) stream_query =
  | Head : ('a, 'a) stream_query
  | Tail : ('a stream, 'a) stream_query
```

This is a GADT: while the second type parameter of the type stream_query uniformly denotes the type of the stream elements, the first one differs depending on the constructor. This type parameter actually encodes the type of the outcome of the requested observation. In this example, the first parameter of the type of *Head* (resp. *Tail*) is $'a$ (resp. $'a$ stream) because observing the head (resp. the tail) of a $'a$ stream must produce an $'a$ (resp. an $'a$ stream).

Now, what would be the translation of the stream codatatype declaration? As our inversion of control turns a codata into a function defined by pattern matching over observation requests, we get the following declaration:

```
and 'a stream = Stream of {
  dispatch : 'o.('o, 'a) stream_query → 'o
}
```

This translation of a codatatype declaration encodes in OCaml the assignment of the following system F type to the constructor *Stream*:

$$\forall\, 'a.(\forall\, 'o.('o, 'a)\ \mathsf{stream\_query} \to 'o) \to 'a\ \mathsf{stream}$$

The internal quantification over the output type o is characteristic of the typing of an inversion of control, similar to the *answer types* introduced in CPS-translated types. However, contrary to CPS-translated programs which have no clue about the actual instantiation of the answer types, a translated codata exactly knows all the possible instantiations of the answer types because they are described by the type of the observation requests. Here is now the translation of the Fibonacci sequence of the introduction:

```
let rec fib : int stream =
  let dispatch : type o.(o, int) stream_query → o = function
    | Head → 0
    | Tail →
      let dispatch : type o.(o, int) stream_query → o = function
        | Head → 1
        | Tail → map2 (+) fib (tail fib)
      in Stream { dispatch }
  in Stream { dispatch }
```

The first obvious difference between the source and the target definitions of the Fibonacci sequence is the presence of two nested pattern matchings in the translated program while only one copattern matching appears in the source program. The first pass of our translation indeed *unnests* full copatterns to decompose them into the *simple* copatterns introduced by Setzer et al. [14] but using a different transformation (Section 4.1).

Then, one may wonder why this translation is well-typed: as said earlier, a pattern matching is well-typed only if all its branches have the same type and here, the branch for *Head* on Line 3 produces an integer while the branch for *Tail* produces a stream! The trick is

that all these branches actually have the same type o and that o, the answer type, is locally refined into distinct types in each branch thanks to the GADT stream_query. Hence, since the type equality "o = int" is valid in the branch for *Head*, the integer literal 0 has type o. Similarly, since the type equality "o = int stream" is available in the branch for *Tail*, the term *Stream* {*dispatch*} also has type o.

The last piece of the puzzle is the encoding of the observation site: why is the source term "*fib*#*Tail*" encoded as "*tail* (*fib*)" in the translated Fibonacci? Let us now reveal the implementation of *tail*:

```
let tail (Stream {dispatch}) = dispatch Tail
```

This piece of code is actually the place where the inversion of control takes place: the observation of the tail of a stream is translated into the evaluation of the *dispatch* function applied to the observation request *Tail*.

As a side note, notice that this translation of *fib* is actually quite inefficient because there is no sharing between the two streams *fib* and *tail* (*fib*). This inefficiency can be avoided using lazy computations as we shall see in Section 4.4.

To conclude this section, let us consider now a slightly more involved example that uses an *indexed* codatatype[15]. Just like a GADT, an indexed codatatype is a codatatype whose parameters encode static information, e.g. an algorithmic invariant. To illustrate this point, let us consider the following declaration for the (codata)type of "fair bistreams of integers":

```
type (_, _) fbs = {
  Left  : int * (read, 'b) fbs     ← (unread, 'b) fbs;
  Right : int * ('a, read) fbs     ← ('a, unread) fbs;
  BTail : (unread, unread) fbs ← (read, read) fbs;
}
```

where read and unread are two incompatible type constructors.

A fair bistream represents two streams of integers that must be consumed in a synchronized way. To enforce this fairness, the *type index* encodes the status of the head of each stream: it can be either read or unread. Then, three observations are possible: (i) the consumer of the stream can choose to read from the *Left* stream to get both an integer and a new bistream whose type witnesses the read status of the left head ; (ii) the consumer can symmetrically choose to read from the *Right* stream ; (iii) the consumer can observe the tails of the two streams but only if it has read the heads of both streams.

Now imagine that the internal state of a bistream is implemented by 2-buffers of the following type:

```
type ('a, 'b, 'e) twobuffer =
  | E :              (read, read, 'e) twobuffer
  | L : 'e       → (unread, read, 'e) twobuffer
  | R : 'e       → (read, unread, 'e) twobuffer
  | F : 'e * 'e → (unread, unread, 'e) twobuffer
```

A 2-buffer has four possible states depending on the order in which its elements are consumed. With this precise state decomposition, as soon as one of the two elements is consumed, it is removed from the buffer to allow the garbage collector to free its memory block. This is a GADT: the status of each element — being read or unread — is encoded in a type parameter. Thanks to type refinements, we can enforce the consistency between such an internal buffer and the observations that have been made. Here is how an iteration over the integers can be represented by a fair bistream:

```
let corec zfrom : type a b.int → (a, b, int) twobuffer → (a, b) fbs with
  | (..n E)#BTail          → zfrom (n + 1) (F (n, −n))
  | (..n (L x))#Left       → (x, zfrom n E)
  | (..n (F (x, y)))#Left  → (x, zfrom n (R y)))
  | (..n (R x))#Right      → (x, zfrom n E)
  | (..n (F (x, y)))#Right → (y, zfrom n (L x))
let z : (unread, read) fbs = zfrom 0 (L 0)
```

The left stream of z represents the positive integers while the right stream of z represents the negative integers. Since we want to allow -0 to be skipped, the type of z already marks the head of the right stream as read. The bistream z is actually defined by an auxiliary recursive function *zfrom*. Its input is an integer n which denotes the depth of the iteration and a buffer *buf*. Its output is a bistream defined by copattern matching. In each branch, a case analysis over *buf* determines the next state of the internal buffer. The type equalities available thanks to the precise typing of the observations allow us to ignore impossible cases. In the first branch, as the typechecker knows from the declaration of *Tail* that "a = read" and "b = read", it can deduce that the only possible case for *buf* is *E*. Indeed, other buffer constructors assume that either a or b is equal to unread. For similar reasons, a buffer constructed with the constructor *R* (resp. *L*) cannot appear in the branch for the *Left* (resp. *Right*) observation.

The translation of this codatatype definition introduces the following GADT for observation queries:

```
type (_, 'a, 'b) fbs_query =
  | Left : (int * (read, 'b) fbs, unread, 'b) fbs_query
  | Right : (int * ('a, read) fbs, 'a, unread) fbs_query
  | BTail : ((unread, unread) fbs, read, read) fbs_query
```

as well as the following functions to perform these queries:

```
(*btail : (read, read) fbs → (unread, unread) fbs*)
let btail (Fbs {dispatch}) = dispatch BTail
(*left : (unread, 'b) fbs → int * (read, 'b) fbs*)
let left (Fbs {dispatch}) = dispatch Left
(*right : ('a, unread) fbs → int * ('a, read) fbs*)
let right (Fbs {dispatch}) = dispatch Right
```

The GADT fbs_query not only captures the dependency between an observation request and the expected type of its outcome, it also *restricts* the domain of the observation *Tail* to the subset of bistreams whose heads have been read and the domain of the observation *Left* (resp. *Right*) to the subset of bistreams whose left (resp. right) head has not been read yet. Finally, the translation of *zfrom* is:

```
let rec zfrom : type a b.int → (a, b, int) twobuffer → (a, b) fbs
 = fun n buf →
   let dispatch : type o.(o, a, b) fbs_query → o = function
     | BTail → (match buf with
       | E → zfrom (n + 1) (F (n, −n)))
     | Left → (match buf with
       | L x → (x, zfrom n E)
       | F (x, y) → (x, zfrom n (R y)))
     | Right → (match buf with
       | R x → (x, zfrom n E)
```

```
    | F (x, y) → (y, zfrom n (L x)))
in Fbs {dispatch}
```

3 FORMALIZATION

In this section, we formalize a source core language λ^{C}, a target core language λ^{G} as well as a translation from λ^{C} to λ^{G} which, in our opinion, captures the essence of our extension of OCaml with copatterns. The section 4 will complete this description with more practical details.

Conventions. We write $\overline{X}$ as a metavariable for a vector of X and $\bullet$ for an empty vector. When a element of a vector must be singularized, we may use comma-based notation like in $\overline{X}, x, \overline{X}'$ or the usual indices-based notation. We often lift judgments or functions from X to $\overline{X}$ without explicit definitions because these definitions are easily deduced from the context. We use the notation $x \# X$ to assert that the name x is not free in X.

3.1 Source language

3.1.1 Syntax. The syntax of the source language is defined in Figure 1. The syntax for variables and applications are standard while the other constructions need some explanations. In previous presentations of calculus with copatterns[1, 15], an observation is written "$t \cdot D$". In λ^{C}, an observation is written "$t \cdot u$". This notation is more general since the *observation request* on t can be computed by an arbitrary term u. Such a term can simply be the literal term D for the most basic observations, but more complex terms, like for instance an application "$t\ u$", can also be used. In other words, this construction makes observation requests first-class citizens of λ^{C}.

In λ^{C}, a data constructor has a single argument contrary to the data constructors of ML which usually accept zero, one or more arguments. As λ^{C} contains codata, there is no loss of generality induced by this rigid syntax: indeed, the argument of the constructor can be of a unit type or of a tuple type, which are definable as codatatypes (see forthcoming Example 3.1).

The language λ^{C} embeds a standard construction for recursive functions of the form "$\mu^{+} f : \sigma := \lambda \overline{x} \{\overline{b}\}$". In that term, the identifiers f and $\overline{x}$ are bound into $\overline{b}$. A recursive function must be annotated by its type scheme σ. In addition, λ^{C} offers syntax to define co-recursive functions of the form "$\mu^{-} f : \sigma := \lambda \overline{x} \{\overline{b}\}$". The distinction between recursive functions and corecursive functions lies in the syntax of their branches: a recursive function defines a computation by pattern matching and recursion while a corecursive function defines a computation by observations and corecursion. Therefore, a recursive pattern matching function makes use of branches of the form "$K\ x \Rightarrow t$" while a corecursive copattern matching function makes use of branches of the form "$\cdot D \Rightarrow t$". A standard λ-abstraction, which by definition eliminates neither a constructor nor an observation, is defined using a single branch of the form "$\bullet \Rightarrow t$" where t is its suspended body. Here $\bullet$ denotes the absence of copattern. Notice that with this syntax, the vector of arguments $\overline{x}$ can be empty: this allows representing recursive data and codata. Abel and Pientka [2] introduce a notion of *Generalized λ-abstraction* which encompasses both recursive and corecursive functions. Nevertheless, maintaining a distinction between these two notions introduced strong invariants which simplified our type system, our translation and our proofs.

The syntax of branches is too atomic to represent nested copatterns like the ones used in the definition of the Fibonacci sequence for instance. As shown by Setzer et al. [14], copattern unnesting

			Terms
t, u	::=	x	*Variable*
	|	D	*Request*
	|	$K\ t$	*Construction*
	|	$t\ t$	*Application*
	|	$t \cdot t$	*Observation*
	|	$\mu^{+} f : \sigma := \lambda \overline{x} \{\overline{b}\}$	*Function*
	|	$\mu^{-} f : \sigma := \lambda \overline{x} \{\overline{b}\}$	*Codata*
			Branches
b	::=	$\bullet \Rightarrow t$	*Suspension*
	|	$\cdot D \Rightarrow t$	*Observation case*
	|	$K\ x \Rightarrow t$	*Deconstruction case*
			Values
v	::=	$\lambda^{-} \overline{x} \{\overline{b}\}$	*Codata*
	|	$\lambda^{+} \overline{x} \{\overline{b}\}$	*Function*
	|	$K\ v$	*Data*
	|	D	*Request*
			Evaluation contexts
E	::=	$[]$	*Hole*
	|	$E\ t$	*On the left of an application*
	|	$v\ E$	*On the right of an application*
	|	$E \cdot v$	*On the left of an observation*
	|	$t \cdot E$	*On the right of an observation*
	|	$K\ E$	*Under construction*
			Types
τ, ρ, ω	::=	α	*Type variable*
	|	$\epsilon^{+}(\overline{\tau})$	*Data*
	|	$\epsilon^{-}(\overline{\tau})$	*Codata*
	|	$\tau \to \tau$	*Arrow*
	|	$\tau \leftarrow \epsilon^{-}(\overline{\tau})$	*Observation requests*
			Type scheme
σ	::=	$\forall \overline{\alpha}.\tau$	
			Typing environments
Γ	::=	$\bullet$	*Empty*
	|	$\Gamma\ \alpha$	*Bind type variable*
	|	$\Gamma\ (x : \sigma)$	*Bind variable*
			Type constraints
C	::=	**true**	*Trivial constraint*
	|	**false**	*Empty constraint*
	|	$\tau = \tau$	*Type equality*
	|	$C \wedge C$	*Conjunction*

Figure 1: Syntax for the source language.

can be performed to automatically transform our definition of the Fibonacci sequence to only contain the simple copatterns of λ^{C}. We discuss our variant of this transformation in Section 4.1.

3.1.2 Semantics. The syntax for values and evaluation contexts is defined in Figure 1. The values include codata, functions, values tagged by a constructor and observation requests. Notice also that values are a subset of terms since a value of the form $\lambda^{\diamond}\overline{x}\{\overline{b}\}$ is a syntactic sugar for $\mu^{\diamond}f : \sigma := \lambda\overline{x}\{\overline{b}\}$ where f is not free in $\overline{b}$ and $\diamond \in \{+, -\}$. The evaluation contexts encode a standard call-by-value weak reduction strategy. To obtain a deterministic semantics, we fix an evaluation order: the evaluation of applications follows a left-to-right order and the evaluation of observations follows right-to-left order. (Other evaluation orders would work.)

The operational semantics for the source language is defined in Figure 2. The rule (SCxt) plugs a reduced term under an evaluation context. The rule (SUnr) replaces by its definition the free occurrences of a (co)recursive function identifier inside its own definition leading to a (co)recursive λ-abstraction value. As long as this λ-abstraction has formal arguments which are variables and if the rule (SEval) is not applicable, the rule (SPush) performs standard call-by-value β-reduction. If the λ-abstraction has only one formal argument x and its first branch is a suspended computation of term t, the free occurrences of x are replaced by the actual argument in t using rule (SEval). If the λ-abstraction is a recursive function with $\overline{x} = \bullet$, then the actual argument must be a constructed value $K\,v$. There are two cases: (i) if K is the constructor used in the pattern $K\,x$ of the first branch, the rule (SDes) applies and the body of the first branch is evaluated under the substitution $x \mapsto v$; (ii) if K is not the constructor of the pattern of the first branch, the rule (SDesF) applies and the evaluation continues with the remaining branches. The rule (SObs) and the rule (SObsF) follow the same shape except that an observation copattern binds no variable, so no substitution is required to proceed to the evaluation of the branch body.

Design choices. Notice that the absence of subcomponents in an observation copattern generates no loss of expressiveness. Indeed, it suffices to have the observations return a function to allow the environment to communicate additional values to the codata.

Alternatively, we could have let observation requests carry a value using a syntax of the form $D(v)$. In that case, as noticed by reviewers of a previous version of this paper, an observation could encode a standard application and λ-abstractions could be represented by a codatatype with a single observation `invoke`. This design would certainly make the source language more minimal and we actually considered it, but we found (i) our translation less elementary in that setting as the introduction of functions of the target language would need special cases and (ii) the resulting source language more difficult to connect with our extension of OCaml.

3.1.3 Type system. The syntax for the types and the typing constraints of the source language is defined in Figure 1. A type can be a type variable α, the application of a type constructor for a GADT $\epsilon^{+}(\bullet)$ or for a codatatype $\epsilon^{-}(\bullet)$, an arrow type $\tau \to \tau$ and a coarrow type for observation requests $\tau \leftarrow \epsilon^{-}(\overline{\tau})$. The reason

$$
\begin{array}{rcl}
E[t] & \overset{\text{SCxt}}{\to} & E[t'] \\
 & & \text{if } t \to t' \\
\mu^{\diamond}f : \sigma := \lambda\overline{x}\{\overline{b}\} & \overset{\text{SUnr}}{\to} & \lambda^{\diamond}\overline{x}\{(\overline{b}[f \mapsto \mu^{\diamond}f : \sigma := \lambda\overline{x}\{\overline{b}\}])\} \\
 & & \text{where } \diamond \in \{+, -\} \\
(\lambda^{\diamond}x\overline{x}\{\overline{b}\})\,v & \overset{\text{SPush}}{\to} & \lambda^{\diamond}\overline{x}\{\overline{b}[x \mapsto v]\} \\
 & & \text{if (SEval) is not applicable} \\
 & & \text{and } \diamond \in \{+, -\} \\
(\lambda^{\diamond}x\{\bullet \Rightarrow t \mid \overline{b}\})\,v & \overset{\text{SEval}}{\to} & t[x \mapsto v] \\
 & & \text{if } \diamond \in \{+, -\} \\
(\lambda^{+} \bullet \{K\,x \Rightarrow t \mid \overline{b}\})\,(K\,v) & \overset{\text{SDes}}{\to} & t[x \mapsto v] \\
(\lambda^{+} \bullet \{K\,x \Rightarrow t \mid \overline{b}\})\,(K'\,v) & \overset{\text{SDesF}}{\to} & (\lambda^{+} \bullet \{\overline{b}\})\,(K'\,v) \\
 & & \text{if } K' \neq K \\
(\lambda^{-} \bullet \{\cdot D \Rightarrow t \mid \overline{b}\})\,D & \overset{\text{SObs}}{\to} & t \\
(\lambda^{-} \bullet \{\cdot D \Rightarrow t \mid \overline{b}\})\,D' & \overset{\text{SObsF}}{\to} & (\lambda^{-} \bullet \{\overline{b}\})\,D' \\
 & & \text{if } D' \neq D
\end{array}
$$

Figure 2: Source small-step operational semantics.

why we choose to assign a type written with a reversed arrow to observation requests will be clear in the forthcoming typing rules.

We assume that type constructors are introduced using toplevel declarations of the form:

$$
\begin{array}{rcl}
\epsilon^{+}(\overline{\alpha}) & := & \Sigma_i K_i : \forall\overline{\alpha}.\tau_i \to \epsilon^{+}(\overline{\tau}_i) \\
\epsilon^{-}(\overline{\alpha}) & := & \times_i D_i : \forall\overline{\alpha}.\tau_i \leftarrow \epsilon^{-}(\overline{\tau}_i)
\end{array}
$$

Example 3.1. The unit type is defined by[1]

$$\text{unit} = \text{DUnit} : \text{unit} \leftarrow \text{unit}$$

and the pair type is defined by

$$\text{pair}(\alpha, \beta) = \begin{cases} \text{DFst} : \forall\alpha\beta.\alpha \leftarrow \text{pair}(\alpha, \beta) \\ \text{DSnd} : \forall\alpha\beta.\beta \leftarrow \text{pair}(\alpha, \beta) \end{cases}$$

Compared to the previous work on indexed codatatypes, our definition may seem less expressive since no type equalities explicitly appear in the declaration of a codatatype. Actually, type equalities are encoded by the fact that the type constructors $\epsilon^{+}(\bullet)$ and $\epsilon^{-}(\bullet)$ are not applied to type variables but to ground types. Besides, since our system includes GADTs, explicit type equalities can be embedded in the type of an observation request by using the usual GADT $\mathsf{eq}(\tau, \rho)$ whose single inhabitant represent the equality between τ and ρ. This GADT can also be used to represent the empty type if it is instantiated with two incompatible ground types, e.g. $\mathsf{eq}(\mathtt{pair}(\mathtt{unit}, \mathtt{unit}), \mathtt{unit})$.

The syntax for typing environments is standard as it allows to introduce type variables and bindings from value identifiers to type schemes. To support GADTs and indexed codatatypes, we also need a syntax for typing constraints C which denote the type equalities available in the current context. These typing constraints include the trivial typing constraint, the empty typing constraint, atomic type equalities and conjunctions of constraints.

The type system for the source language is defined in Figure 3. It is specified by two judgments: "$\Gamma, C \vdash t : \tau$" is read "Under

[1] Any definition would work as soon as it is isomorphic to a singleton type.

the typing environment Γ and the typing constraint C, the term t has (the monomorphic) type τ." and "$\Gamma, C \vdash \lambda\overline{x}.\overline{b} : \diamond\tau$" is read "Under the typing environment Γ and the typing constraint C, the abstraction $\lambda\overline{x}.\overline{b}$ is the body of $\diamond$-function and has type τ." where $\diamond \in \{+, -\}$.

Notice that there is no judgment for polymorphic terms since type schemes are immediately instantiated to the required monomorphic type as in standard syntax directed presentations of ML. The rule (SConv) uses an auxiliary judgment of the form $C \Vdash \tau = \rho$ which is read "The typing constraint C implies that the type τ is equivalent to type ρ".

The rules (SVar) and (SApply) are standard. The rule (SRequest) instantiates the type scheme of the considered observation request D to comply with the expected type. The rule (SConstruct) instantiates the type scheme of the considered constructor K to match both the type of its argument and the expected type. The rule (SObserve) is the dual of the rule (SApply): it checks that its left-hand-side has the type expected by the observation request on its right-hand-side. The fact that the reversed arrow is on the side of the observation request, not the codata, is the reason why we consider that the observation request term is the one that is *logically driving* the computation, even though the actual code is contained in the codata definition.

The rule (SFun) checks that a recursive or corecursive function is well-formed. It is parameterized by the nature of the function $\diamond \in \{+, -\}$. The user type annotation $\forall\overline{\alpha}.\rho$ is assigned to the function identifier f in the typing environment to be able to type (co)recursive occurrence of f in its body. Verifying that the definition of f actually enjoys the programmer type ascription is the responsibility of the auxiliary typing judgment "$\Gamma\,(f : \forall\overline{\alpha}.\rho)\overline{\alpha}, C \vdash \lambda\overline{x}.\overline{b} : \diamond\rho$". Notice that $\diamond$ is transmitted to this auxiliary judgment because its rules are specific to the nature of the function being checked. The rule (SLam) is a standard type checking rule for λ-abstractions . Once all the λ-abstractions are checked, (SPat) applies if $\diamond = +$ and (SCoPat) applies if $\diamond = -$.

The rule (SPat) is a standard type checking rule for pattern matching in presence of GADT: the type scheme of the constructor K is used to determine the type of the variable x introduced by the pattern K . The input type of the expected arrow type must share the same type constructor as the data constructor's. Yet, their arguments may differ and this difference is exactly the type refinement introduced by the constructor pattern in the branch body. For this reason, the equalities of type arguments, i.e. $\overline{\tau} = \overline{\tau}'$, are locally assumed in C to check t. Finally, the local type variables $\overline{\beta}$ are introduced in the context. Being local, they cannot escape their scope as enforced by the hypothesis $\overline{\beta} \# \Gamma, C, \tau$. The rule (SCoPat) is dedicated to the typechecking of copattern matchings. In that case, the expected type is not an arrow but a codatatype, which is consistent with the rule (SObserve). The rule uses a similar technique as (SPat) to extract type refinements except that now, that is the input type — not the output type — of the coarrow type of D which is confronted with the expected type to determine the available type equalities. The output type τ of D type scheme is the type of the observation outcome. Thus, the rule must check that the branch body t is of that type. Again, a freshness hypothesis ensures that the local type variables $\overline{\beta}$ do not escape their scope.

$$\boxed{\Gamma, C \vdash t : \tau}$$

(SConv)
$$\frac{\Gamma, C \vdash t : \tau \qquad C \Vdash \tau = \rho}{\Gamma, C \vdash t : \rho}$$

(SVar)
$$\frac{\Gamma(x) \preceq \tau}{\Gamma, C \vdash x : \tau}$$

(SRequest)
$$\frac{D \preceq \tau \leftarrow \rho}{\Gamma, C \vdash D : \tau \leftarrow \rho}$$

(SConstruct)
$$\frac{K \preceq \rho \to \tau \qquad \Gamma, C \vdash t : \rho}{\Gamma, C \vdash K\,t : \tau}$$

(SObserve)
$$\frac{\Gamma, C \vdash t : \tau \qquad \Gamma, C \vdash u : \rho \leftarrow \tau}{\Gamma, C \vdash t \cdot u : \rho}$$

(SApply)
$$\frac{\Gamma, C \vdash t : \tau \to \rho \qquad \Gamma, C \vdash u : \tau}{\Gamma, C \vdash t\,u : \rho}$$

(SFun)
$$\frac{\forall\overline{\alpha}.\rho \preceq \tau \qquad \Gamma\,(f : \forall\overline{\alpha}.\rho)\overline{\alpha}, C \vdash \lambda\overline{x}.\overline{b} : \diamond\,\rho}{\Gamma, C \vdash \mu^{\diamond} f : \forall\overline{\alpha}.\rho := \lambda\overline{x}\{\overline{b}\} : \tau}$$

$$\boxed{\Gamma, C \vdash \lambda\overline{x}.\overline{b} : \diamond\,\tau}$$

(SPat)
$$\frac{\begin{array}{c}\overline{\beta} \# \Gamma, C, \epsilon^{+}(\overline{\tau}), \tau \\ K :: \forall\overline{\beta}.\tau_x \to \epsilon^{+}(\overline{\tau}') \\ \Gamma\,\overline{\beta}\,(x : \tau_x), C \wedge \overline{\tau} = \overline{\tau}' \vdash t : \tau \\ \Gamma, C \vdash \lambda\bullet\{\overline{b}\} :+\ \epsilon^{+}(\overline{\tau}) \to \tau\end{array}}{\Gamma, C \vdash \lambda\bullet\{K\,x \Rightarrow t \mid \overline{b}\} :+\ \epsilon^{+}(\overline{\tau}) \to \tau}$$

(SCoPat)
$$\frac{\begin{array}{c}\overline{\beta} \# \Gamma, C, \epsilon^{-}(\overline{\tau}) \\ D :: \forall\overline{\beta}.\tau \leftarrow \epsilon^{-}(\overline{\tau}') \\ \Gamma\,\overline{\beta}, C \wedge \overline{\tau} = \overline{\tau}' \vdash t : \tau \\ \Gamma, C \vdash \lambda\bullet\{\overline{b}\} :-\ \epsilon^{-}(\overline{\tau})\end{array}}{\Gamma, C \vdash \lambda\bullet\{\cdot D \Rightarrow t \mid \overline{b}\} :-\ \epsilon^{-}(\overline{\tau})}$$

(SLam)
$$\frac{\Gamma\,(x : \tau), C \vdash \lambda\overline{x}.\overline{b} : \diamond\rho}{\Gamma, C \vdash \lambda x\overline{x}.\overline{b} : \diamond\,\tau \to \rho}$$

Figure 3: Type system for the source language

We are now ready to state the type soundness for this system in the form of the standard Subject Reduction and Progress theorems.

Theorem 3.2 (Subject reduction for the source language). *Let C be satisfiable. If $\Gamma, C \vdash t : \tau$ holds and $t \to t'$, then $\Gamma, C \vdash t' : \tau$.*

Theorem 3.3 (Progress for the source language). *Let C be satisfiable. If $\Gamma, C \vdash t : \tau$ holds, then either t is a value or there exists t' such that $t \to t'$.*

One important corollary of subject reduction is the fact that if some code happens to live under an inconsistent constraint C then this code is dead, i.e. unreachable by the evaluation.

3.2 Target language

3.2.1 Syntax. The syntax of the target language λ^{G} is defined in Figure 4. It differs from the source language λ^{C} on two points: first, there are no constructions to denote observations, observation requests and copattern matching ; second, data constructors and constructor patterns now accepts zero, one or several arguments. To simplify our presentation, we informally assume that λ^{G} has

			Terms
t, u	::=	x	*Variable*
	|	$K\,\overline{t}$	*Construction*
	|	$t\,t$	*Application*
	|	$\mu^+ f : \sigma := \lambda\overline{x}.\overline{b}$	*Function*
			Branches
b	::=	$\bullet \Rightarrow t$	*Suspension*
	|	$K\,\overline{x} \Rightarrow t$	*Deconstruction case*
			Values
v	::=	$\lambda\overline{x}.\overline{b}$	*Function*
	|	$K\,\overline{v}$	*Data*
			Evaluation contexts
E	::=	$[]$	*Hole*
	|	$E\,t$	*On the left of an application*
	|	$v\,E$	*On the right of an application*
	|	$K\,(\overline{v}, E, \overline{t})$	*Under construction*
			Types
τ, ρ, ω	::=	α	*Type variable*
	|	$\epsilon^+(\overline{\tau})$	*Data*
	|	$\tau \to \tau$	*Arrow*
			Type scheme
σ	::=	$\forall\overline{\alpha}.\tau$	
			Typing environments
Γ	::=	$\bullet$	*Empty*
	|	$\Gamma\,\alpha$	*Bind type variable*
	|	$\Gamma\,(x : \sigma)$	*Bind variable*
			Type constraints
C	::=	**true**	*Trivial constraint*
	|	**false**	*Empty constraint*
	|	$\tau = \tau$	*Type equality*
	|	$C \wedge C$	*Conjunction*

Figure 4: Syntax for the target language.

a notion of toplevel ground value definition, i.e. a value with no free variables. We will write: "$x := t$" to denote such a toplevel definition.

3.2.2 Semantics. The syntax for values and evaluation contexts of the target language is defined in Figure 4. Codata values and observation-related evaluation contexts have been removed from the values and evaluations contexts of λ^{C}. As for the syntax of terms, a data constructor of λ^{G} accepts a possibly empty vector of values and these values are produced from left-to-right as witnesses by the production named (UnderConstruction) of the grammar for evaluation contexts. The operational semantics for the target language is defined in Figure 5. Again, it is made of almost the same rules as the operational semantics of λ^{C} except that the rules for observations are removed and that the reduction of a pattern matching branch now binds not necessarily one variable but a possibly empty vector of variables.

$$
\begin{array}{rcl}
E[t] & \overset{\text{(TCxt)}}{\longrightarrow} & E[t'] \\
 & & \text{if } t \longrightarrow t' \\
\mu^+ f : \sigma := \lambda\overline{x}.\overline{b} & \overset{\text{(TUnr)}}{\longrightarrow} & \lambda\overline{x}.(\overline{b}[f \mapsto \mu^+ f : \sigma := \lambda\overline{x}.\overline{b}]) \\
(\lambda x\overline{x}.\overline{b})\,v & \overset{\text{(TPush)}}{\longrightarrow} & \lambda\overline{x}.\overline{b}[x \mapsto v] \\
 & & \text{if (TEval) does not apply.} \\
(\lambda x.\bullet \Rightarrow t \mid \overline{b})\,v & \overset{\text{(TEval)}}{\longrightarrow} & t[x \mapsto v] \\
(\lambda \bullet .K\,\overline{x} \Rightarrow t \mid \overline{b})\,(K\,\overline{v}) & \overset{\text{(TDes)}}{\longrightarrow} & t[\overline{x} \mapsto \overline{v}] \\
(\lambda \bullet .K\,\overline{x} \Rightarrow t \mid \overline{b})\,(K'\,\overline{v}) & \overset{\text{(TDesF)}}{\longrightarrow} & (\lambda \bullet .\overline{b})\,(K'\,\overline{v}) \\
 & & \text{if } K' \neq K
\end{array}
$$

Figure 5: Target small-step operational semantics.

3.2.3 Type system. The syntax for the types and the typing constraints of the target language is defined in Figure 4. To reflect the change on the syntax for terms, there is no more coarrows and codatatype constructors in the type algebra of λ^{G}. We assume that type constructors are introduced by toplevel declarations of the form:

$$\epsilon^+(\overline{\alpha}) \quad := \quad \Sigma_i K_i : \forall\overline{\alpha}.\overline{\sigma} \to \epsilon^+(\overline{\tau}_i)$$

Notice that the argument of the constructor can have a polymorphic type. Notice also that the type scheme assigned to K is not an ML type scheme σ but a second-order polymorphic type scheme.

The type system for the target language is defined in Figure 6. As in λ^{C}, the type system is based on two judgments: one for terms and another for abstractions. A major difference between the type system of λ^{C} and the type system of λ^{G} is the fact that the judgment for terms assigns a *type scheme* (not a monomorphic type) to a term. As in ML though, many rules only accept monomorphic types to maintain the decidability of type inference. This type system captures the treatment of second order polymorphic types in the OCaml typechecker[6]. The rules (TConv), (TApply), (TRecfun) and (TLam) are similar to their λ^{C} counterparts. The rule (SVar) has been split into two rules (TVar) and (TInst). This separation allows polymorphic identifiers to be extracted as is from the environment to the places where a polymorphic term is expected. (This contrasts with usual syntax-directed typing rules of ML where polymorphic identifiers are immediately instantiated when extracted from the environment.) As specified by (TConstructor), the arguments of a data constructor are such a place: they are assigned the type schemes declared for the arguments of this constructor. In exchange to this harder constraint on the generality of constructor argument types, the variables bound by a pattern can be polymorphic as expressed by the new hypothesis in the rule (TPat) which now introduces $(\overline{x} : \overline{\sigma})$ in the typing environment used to typecheck the branch body. Forgetting about this change, the rule (TPat) is the same as (SPat).

Theorem 3.4 (Subject reduction for the target language). *Let C be satisfiable. If $\Gamma, C \vdash t : \tau$ holds and $t \longrightarrow t'$, then $\Gamma, C \vdash t' : \tau$*

Theorem 3.5 (Progress for the source language). *Let C be satisfiable. If $\Gamma, C \vdash t : \tau$ holds, then either t is a value or there exists t' such that $t \longrightarrow t'$.*

$$\boxed{\Gamma, C \vdash t : \sigma}$$

$$\text{(TConv)}\ \frac{\Gamma, C \vdash t : \tau \qquad C \Vdash \tau = \rho}{\Gamma, C \vdash t : \rho} \qquad \text{(TVar)}\ \frac{\Gamma(x) = \sigma}{\Gamma, C \vdash x : \sigma}$$

$$\text{(TInst)}\ \frac{\Gamma, C \vdash x : \sigma' \qquad \sigma' \preceq \sigma}{\Gamma, C \vdash x : \sigma} \qquad \text{(TConstruct)}\ \frac{K \preceq \overline{\sigma} \to \tau \qquad \Gamma, C \vdash \overline{t} : \overline{\sigma}}{\Gamma, C \vdash K\,\overline{t} : \tau}$$

$$\text{(TApply)}\ \frac{\Gamma, C \vdash t : \tau \to \rho \qquad \Gamma, C \vdash u : \tau}{\Gamma, C \vdash t\,u : \rho} \qquad \text{(TRecfun)}\ \frac{\Gamma\,(f : \forall\overline{\alpha}.\rho)\overline{\alpha}, C \vdash \lambda\overline{x}.\overline{b} : \rho}{\Gamma, C \vdash \mu^{+} f : \forall\overline{\alpha}.\rho := \lambda\overline{x}.\overline{b} : \forall\overline{\alpha}.\rho}$$

$$\boxed{\Gamma, C \vdash \lambda\overline{x}.\overline{b} : \tau}$$

$$\text{(TLam)}\ \frac{\Gamma\,(x : \tau), C \vdash \lambda\overline{x}.\overline{b} : \rho}{\Gamma, C \vdash \lambda x\overline{x}.\overline{b} : \tau \to \rho} \qquad \text{(TPat)}\ \frac{\begin{array}{c}\overline{\beta} \# \Gamma, \tau, \epsilon^{+}(\overline{\tau}) \\ K :: \forall\overline{\beta}.\overline{\sigma} \to \epsilon^{+}(\overline{\tau}') \\ \Gamma\,\overline{\beta}\,(\overline{x} : \overline{\sigma}), C \wedge \overline{\tau} = \overline{\tau}' \vdash t : \tau \\ \Gamma, C \vdash \lambda\bullet\{\overline{b}\} : \epsilon^{+}(\overline{\tau}) \to \tau\end{array}}{\Gamma, C \vdash \lambda\bullet\{K\,\overline{x} \Rightarrow t \mid \overline{b}\} : \epsilon^{+}(\overline{\tau}) \to \tau}$$

Figure 6: Type system for the target language.

3.3 Translation

As the source and the target languages are set up, we have everything we need to describe the translation specified by Figure 7. The translation is made of three components: a translation of terms, a translation of types and a translation of type declarations.

The translation of types is the identity for type variables and is simply going through the structure of arrow types and datatypes. A coarrow type of the form $\tau \leftarrow \epsilon^{-}(\overline{\tau})$ is rewritten into an instantiation of a type constructor $\epsilon_r^{-}([\![\tau]\!][\![\rho]\!])$. As we shall see, this type constructor is introduced by the translation of type declarations and is the GADT for reified observation requests. A codatatype $\epsilon^{-}(\overline{\tau})$ is rewritten by simply changing the type constructor $\epsilon^{-}(\bullet)$ to $\epsilon_d^{-}(\bullet)$ which is also introduced by the translation of type declarations.

On datatype declarations, the translation only rewrites the types that occur in data constructor type schemes. On each codatatype declaration, the translation creates three toplevel declarations:

(1) A GADT declaration for reified observation requests $\epsilon_r^{-}(\overline{\alpha}) := \Sigma_i D_i : \forall\overline{\alpha}.\epsilon_r^{-}([\![\tau_i]\!], [\![\overline{\tau}_i]\!])$. where τ_i is the (source) type of the outcome of this observation and $\overline{\tau}_i$ is the indices specified for D_i in its declaration in the source program.
(2) Another datatype declaration to type translated codata $\epsilon_d^{-}(\overline{\alpha}) := K^{\epsilon^{-}(\overline{\alpha})} : \forall\overline{\alpha}.(\forall\beta.\epsilon_r^{-}(\beta, \overline{\alpha}) \to \beta) \to \epsilon_d^{-}(\overline{\alpha})$. The argument of $K^{\epsilon^{-}(\overline{\alpha})}$ is a *polymorphic dispatch function* parameterized by the type of the outcome. This function takes as input an observation requests and returns as output the outcome of this observation.
(3) A set of toplevel value definitions for the higher-order translation of the observation requests $(D_i)_{i \in I}$. We use the convention that for each D_i, there is a related function d_i such that $d_i := \mu^{+}\_ : \forall\overline{\alpha}.\epsilon_d^{-}([\![\overline{\tau}_i]\!]) \to [\![\tau_i]\!] := \lambda\bullet.K^{\epsilon^{-}(\overline{\alpha})}\,c \Rightarrow c\,D_i$. This function extracts the polymorphic dispatch function out of the translated codata and it applies this function to the reified observation requests D_i.

At the typing level, the observation functions d_i translate the source type of each D_i *by reversing* their arrows:

Lemma 3.6. *If $D_i : \forall\overline{\alpha}.\tau_i \leftarrow \epsilon^{-}(\overline{\tau}_i)$ in the source program, then $d_i : \forall\overline{\alpha}.\epsilon^{-}([\![\overline{\tau}_i]\!]) \to [\![\tau_i]\!]$ in the target program.*

As a consequence, in the translation for terms, the observation $t \cdot u$ is rewritten by simply reversing the application order into $[\![u]\!]\,[\![t]\!]$ and an observation query D is simply replaced by its lowercase version d. As directed by the translation of types, a corecursive function is translated into an application of K constructor to the translation $[\![\bigvee_i \cdot D_i \Rightarrow t]\!]^{\bot}_{\overline{\alpha}, \epsilon^{-}(\overline{\tau})}$ of the copattern matching, which is now a pattern matching over the reified observations. For the other cases, namely (R-Var), (R-Apply), (R-Construct), (R-Data) and (R-PatBranch), the translation goes through the source terms without changing their head structure. We extend the translation of types to a translation of typing constraints and typing environments in a natural way. Then, we can state that the translation is type preserving.

Theorem 3.7 (Type preservation). *Let C be satisfiable. If $\Gamma, C \vdash t : \tau$ holds, then $[\![\Gamma]\!], [\![C]\!] \vdash [\![t]\!] : [\![\tau]\!]$ holds.*

The translated code simulates the source code. The simulation is based on a one-to-many relation.

Theorem 3.8 (Simulation). *If $t \to t'$, then $[\![t]\!] \to^{+} [\![t']\!]$.*

4 IMPLEMENTATION

In this section, we give some insights about our proof-of-concept implementation. This prototype is a fork of the official compiler available using the `opam` package manager with the following command: `opam switch 4.04.0+copatterns`.

4.1 Unnesting of copatterns

The actual syntax for copatterns in our extension allows copatterns and patterns to be nested:

			Patterns
p	::=	x	*Variable pattern*
	|	$K\,p$	*Constructor pattern*
			Copatterns
q	::=	$\cdot$	*Hole*
	|	$q \cdot D$	*Destructor copattern*
	|	$q\,p$	*Application copattern*

As shown by Setzer et al. [14], a nested copattern matching can be rewritten to an equivalent copattern matching in which each nested copattern has been unnested thanks to the introduction of simple copattern matchings. The main idea of this compilation scheme consists in decomposing a chain of destructors into several copattern matchings while passing the intermediate patterns

Types

$$
\begin{array}{rcl}
[\![\alpha]\!] & = & \alpha \\
[\![\tau \to \rho]\!] & = & [\![\tau]\!] \to [\![\rho]\!] \\
[\![\epsilon^+(\overline{\tau})]\!] & = & \epsilon^+([\![\overline{\tau}]\!]) \\
[\![\tau \leftarrow \epsilon^-(\overline{\tau})]\!] & = & \epsilon^-_r([\![\tau]\!][\![\overline{\tau}]\!]) \\
[\![\epsilon^-(\overline{\tau})]\!] & = & \epsilon^-_d([\![\overline{\tau}]\!])
\end{array}
$$

Terms

$$
\begin{array}{rcl}
[\![x]\!] & = & x \\
[\![t\ u]\!] & = & [\![t]\!]\ [\![u]\!] \\
[\![t \cdot u]\!] & = & [\![u]\!]\ [\![t]\!] \\
[\![K\ t]\!] & = & K\ [\![t]\!] \\
[\![D]\!] & = & d \\
[\![\mu^- f : \forall \overline{\alpha}.\tau := \lambda \overline{x}\{\overline{b}\}]\!] & = & \mu^+ f : \forall \overline{\alpha}.[\![\tau]\!] := \lambda \overline{x}.K^{\epsilon^-(\overline{\tau})}\ [\![\overline{b}]\!]^{\perp}_{\overline{\alpha}, O(\tau)} \\
[\![\mu^+ f : \forall \overline{\alpha}.\tau := \lambda \overline{x}\{\overline{b}\}]\!] & = & \mu^+ f : \forall \overline{\alpha}.[\![\tau]\!] := \lambda \overline{x}.[\![\overline{b}]\!] \\
[\![K\ x \Rightarrow t]\!] & = & K\ x \Rightarrow [\![t]\!] \\
[\![\bigvee_i \cdot D_i \Rightarrow t]\!]^{\perp}_{\overline{\alpha}, \epsilon^-(\overline{\tau})} & = & \mu^+\_ : \forall \beta.\epsilon^-_r(\beta, \overline{\tau}) \to \beta := \lambda \bullet .\ \bigvee_i D_i \Rightarrow [\![t]\!]
\end{array}
$$

Type declarations

$$
\begin{array}{rcl}
[\![\epsilon^+(\overline{\alpha}) := \Sigma_i K_i : \forall \overline{\alpha}.\tau_i \to \epsilon^+(\overline{\tau}_i)]\!] & = & \epsilon^+(\overline{\alpha}) := \Sigma_i K_i : \forall \overline{\alpha}.[\![\tau_i]\!] \to \epsilon^+([\![\overline{\tau}]\!]_i) \\
[\![\epsilon^-(\overline{\alpha}) := \times_i D_i : \forall \overline{\alpha}.\tau_i \leftarrow \epsilon^-(\overline{\tau}_i)]\!] & = & \left\{ \begin{array}{l}
\epsilon^-_r(\overline{\alpha}) := \Sigma_i D_i : \forall \overline{\alpha}.\epsilon^-_r([\![\tau_i]\!], [\![\overline{\tau}_i]\!]) \\
\epsilon^-_d(\overline{\alpha}) := K^{\epsilon^-(\overline{\alpha})} : \forall \overline{\alpha}.(\forall \beta.\epsilon^-_r(\beta, \overline{\alpha}) \to \beta) \to \epsilon^-_d(\overline{\alpha}) \\
\forall i, d_i := \mu^+\_ : \forall \overline{\alpha}.\epsilon^-_d([\![\overline{\tau}_i]\!]) \to [\![\tau_i]\!] := \lambda \bullet .K^{\epsilon^-(\overline{\alpha})}\ c \Rightarrow c\ D_i
\end{array} \right.
\end{array}
$$

Figure 7: Translation from the source to the target language.

as arguments to auxiliary functions that will drive the algorithm. Consider the following declarations of repr and qrepr.

```
type _ repr =                          type _ qrepr = {
  | Int  : int  → int repr               QInt  : int  ← int qrepr;
  | Bool : bool → bool repr              QBool : bool ← bool qrepr }
```

Now, let us transform the following nested copattern matching.

```
let corec f : type a.a repr → a qrepr with
  | (..(Int n))#QInt  → n
  | (..(Bool b))#QBool → b
```

Following Setzer et al. [14], we start by introducing a fresh "splitting" variable x. Unlike the original algorithm, which delegates the case-splitting on x to auxiliary functions, we accumulate the set of patterns to match on and continue the unnesting in the corresponding branch such that the type equalities introduced by the clause are not lost.

```
let f : type a.a repr → a qrepr = fun x →
  let dispatch : type o.(o, a) qrepr_query → o = function
    | QInt → (match x with Int n → n)
    | QBool → (match x with Bool b → b)
  in Qrepr { dispatch }
```

Also, the unnesting of Setzer et al. [14] considers the copattern matching branches as a *set* of rules whereas our unnesting transformation considers the branches as a *sequence* of rules. In our opinion, this design choice is closer to the standard semantics of ML pattern matching. Consequently, we use a different notion of overlapping than the one found in previous work. This is the topic of the next section.

4.2 Overlapping

The rules of the operational semantics of Section 3 are deterministic even in case of copatterns overlapping. Indeed, the order of definition of rules in the copattern matching defines an order of precedence. For instance, the program:

```
let corec zeros : int stream with
  | ..#Head → 0
  | ..#Head → 1
  | ..#Tail → zeros
```

produces the following warning since the pattern matching of the *dispatch* function inherits from the same redundancy in its definition.

```
  | ..#Head → 1
    ^^^^^^^^^
```

Warning : this match case is unused.

A more complex problem arises in presence of nested copatterns. Consider the following definition where the second rule overlaps with the third and the fourth rules.

```
let corec f : int → int stream with
  | (..n)#Head → 0
  | (..n)#Tail → f (n − 1)
  | (..n)#Tail : int stream with
  | (..n)#Tail#Head → n
  | (..n)#Tail#Tail → f (n + 1)
```

A possible disambiguation of this overlapping would be to strictly enforce the precedence defined by the order of definition. From that point of view, we have that "$f\#Tail\#Tail = f\ (n-1)\#Tail$" and the third and fourth rules are marked as redundant with respect to the second rule.

We decided to refine the disambiguation rule of overlapping considering the deepest rules as the stronger ones: a copattern q is marked as redundant with respect to another copattern q' if q is a prefix of q'. Thus, the second copattern "$f\#Tail$" of our example is marked as redundant because it is a prefix of the last ones.

Intuitively, when the programmer is starting a copattern matching with a copattern of the form "$(f\ n)\#Tail\#Head$", he is expressing the intent of defining the stream "$(f\ n)\#Tail\#Tail$" in the current copattern matching. Therefore a subsequent copattern "$(f\ n)\#Tail$" must be marked as hidden by the first copattern. On the contrary, a subsequent copattern of the form "$(f\ n)\#Tail\#Tail$" is the continuation of the definition of the stream "$(f\ n)\#Tail$" started by the first copattern so it does not make sense to ignore it.

4.3 Coverage

The type soundness theorem of Section 3 assumes that all the copattern matchings are complete: a codata must be able to answer any observation request compatible with its type. In presence of indexed codatatypes, we have seen that the verification of the coverage amounts to complete the copattern matching with the potentially missing cases of observation requests and to prove that the types of the observation requests of these cases are incompatible with the current typing context. Our implementation does not perform this check explicitly since the coverage of a translated copattern matching is equivalent to the exhaustiveness of the *dispatch* function of its translation. Hence, we simply reuse the exhaustiveness checker for GADTs already implemented in the OCaml type checker.

4.4 Lazy evaluation

Let us come back on the translation of the Fibonacci of the Section 2:

```
let rec fib : int stream =
  let dispatch : type o.(o, int) stream_query → o = function
    | Head → 0
    | Tail →
      let dispatch : type o.(o, int) stream_query → o = function
        | Head → 1
        | Tail → map2 (+) fib (tail fib)
      in Stream {dispatch}
  in Stream {dispatch}
```

As is, the execution of this program takes an exponential time because the evaluation of $fib\#Tail^{n+2}\#Head$ requires the evaluation of both $fib\#Tail^{n+1}\#Head$ and $fib\#Tail^{n}\#Head$. However, the stream *fib* is purely functional: there is no need to compute the same observation twice as its outcome will never change.

Relying on OCaml support for laziness, we introduce a new feature for copattern matching: when the programmer needs such sharing, she can prepend **cofunction** with the **lazy** keyword. As shown with the next example, this makes the translation automatically introduce lazy computations.

Assuming that *map2* was also defined with the **lazy** construct, one can write *fib* as below.

```
let rec fib : int stream = lazy cofunction : int stream with
  | ..#Head → 0
  | ..#Tail : int stream with
  | ..#Tail#Head → 1
  | ..#Tail#Tail → map2 (+) fib fib#Tail
```

Our translation produces a program of the following form:

```
let rec fib : int stream =
  let dispatch : type o.(o, int) stream_query → o = function
    | Head → 0
    | Tail →
      let dispatch : type o.(o, int) stream_query → o = function
        | Head → 1
        | Tail → map2 (+) fib (tail fib)
      in
      let hb2 = lazy (dispatch Head) in
      let tb2 = lazy (dispatch Tail) in
      let mem2 : type o.(o, int) stream_query → o = function
        | Head → Lazy.force hb2
        | Tail → Lazy.force tb2
      in Stream {dispatch = mem2}
  in
  let hb1 = lazy (dispatch Head) in
  let tb1 = lazy (dispatch Tail) in
  let mem1 : type o.(o, int) stream_query → o = function
    | Head → Lazy.force hb1
    | Tail → Lazy.force tb1
  in Stream {dispatch = mem1}
```

To transform the computation triggered by observation to lazy computation, intermediate thunks (*hb1*, *hb2*, *tb1* and *tb2*) and *dispatch* memoization functions (*mem1* and *mem2*) are generated. In *mem1* and *mem2*, the right-hand side of each branch is replaced by a call to the *Lazy.force* primitive of OCaml which forces the appropriate thunk to produce a value.

The lazy evaluation produces an in-place memoization of the computation: the first time the computation is forced, an actual evaluation is run. Its outcome is stored in place of the computation itself so that it can be reused in subsequent evaluation. As expected, in the case of the Fibonacci sequence, this optimization restores the linear complexity.

4.5 First-order and higher-order first-class observations

By design, our preprocessor gives the programmer access to the types, values and constructors introduced by the translation. As a consequence, observations of codata can be expressed using different flavors, depending on the level of abstraction and the mechanisms needed by the programmer.

The programmer may be interested in defining his own *derived observations* by composing primitive observations. As usual, higher-order representations of observation requests enjoy a good composability. For instance, defining the observation of the n-th element of a stream s is as simple as follows:

```
let nth n s = head (iterate tail s !! n)
```

The observations-as-functions approach has one disadvantage: closures cannot be compared and used as the keys of an hashtable for instance. If the programmer is interested in that kind of low-level manipulations she has access to the constructors of the observation queries, such as *Head* or *Tail* for stream, which are first-order data, comparable and efficiently hashable. We shall see in Section 5 that this possibility can open new opportunities for the programmer.

5 GAME OF LIFE USING COMONADS

Comonads[9] are dual to monads. To quote Conal Eliott's blog[5], "comonads describe values in [a potentially infinite][1] context". The type of *cobind* is also an illustration of this slogan:

```
type a b . (a comonad → b) → a comonad → b comonad
```

The *cobind* combinator uniformly lifts a local computation represented by a function f of type $a\ comonad \to b$, morally defined on a specific point of type a, into a global, infinite, computation over the entire context of this point. This slogan is made explicit with the following comonad instance: an infinite zipper[7] composed of a cursor, *Proj*, and two streams, *Left* and *Right*. Such zippers are naturally expressed in terms of observations:

```
type 'a zipper = {
  Left  : 'a stream;
  Proj  : 'a;
  Right : 'a stream;
}
```

The implementation of *cobind* for zippers corresponds to an infinite structure whose left and right contexts are defined by a *map*-like function which infinitely pushes f to the left and the right directions:

```
let cobind : type a b . (a zipper → b) → a zipper → b zipper
 = fun f z → cofunction : b zipper with
   | ..#Left  → map_iterate f move_left (move_left z)
   | ..#Proj  → f z
   | ..#Right → map_iterate f move_right (move_right z)
```

The *Left* observation returns a stream of infinite zippers where each of these zippers are an instance of the zipper z with its focus moved a finite number of times to the left and the function f applied to the element under focus. The case for *Right* observation is similar. The *map_iterate* function is straightforward:

```
let rec map_iterate : type a b.(a → b) → (a → a) → a → b stream
  = fun f shift z → cofunction : b stream with
  | ..#Head → f z
  | ..#Tail  → map_iterate f shift (shift z)
```

With the standard copatterns, the *move_left* and *move_right* could be implemented as follows:

```
let move_left : type a . a zipper → a zipper
  = fun z → cofunction : a zipper with
  | ..#Left → z#Left#Tail
  | ..#Proj → z#Left#Head
  | ..#Right : a stream with
  | ..#Right#Head → z#Proj
  | ..#Right#Tail → z#Right
let move_right : type a . a zipper → a zipper
  = fun z → cofunction : a zipper with
  | ..#Left : a stream with
  | ..#Left#Head → z#Proj
  | ..#Left#Tail → z#Left
  | ..#Proj → z#Right#Head
  | ..#Right → z#Right#Tail
```

[1]We added the words between brackets to be a little bit more specific.

...but a programmer would not like it. Indeed, the definitions of *move_right* and *move_left* are redundant and should be factorized. As a matter of fact, there is a dependency between the name of the function and its copatterns but this dependency cannot be captured by the original approach of copatterns. Fortunately, since our observations are reified as first-order data, the following user-defined *dispatch* function expresses the factorization mentioned above:

```
let custom_dispatch : type a.
  ((a, a) zipper_query → a) →
  ((a stream, a) zipper_query → a stream) →
  a zipper
  = fun pr dir → cofunction : a zipper with
    | ..#Proj  → pr Proj
    | ..#Left  → dir Left
    | ..#Right → dir Right
let move : type a.
  (a stream, a) zipper_query →
  (a zipper → a stream) → (a zipper → a stream) →
  a zipper → a zipper
  = fun dir fwd bwd z →
  let corec bs : a stream with
    | ..#Head → z#Proj
    | ..#Tail → bwd z
  in
  custom_dispatch
    (fun _ → (fwd z)#Head)
    (fun dir' → if dir = dir' then (fwd z)#Tail else bs)
```

Then, *move_left* and *move_right* are one-liners:

```
let move_left z = move Left left right z
let move_right z = move Right right left z
```

A nice application of this comonadic zipper is the definition of a one-dimensional Game of Life[3, 13]. A grid in that variation of this game is a doubly-infinite stream of booleans, i.e. a pair of two boolean streams: *true* means that the cell is alive, *false* that it is dead. As the Game of Life is a typical example of local rules that must be applied uniformly to the entire space, a comonadic zipper is an adequate codata structure to implement this game.

A rule is a function of type bool zipper → bool. For instance, we can decide that a cell is alive by following this rule:

```
let rule z =
  let left = (move_left z)#Proj
  and middle = z#Proj
  and right = (move_right z)#Proj
  in ¬ (left ∧ middle ∧ ¬ right ∨ left = middle)
```

To initialize the space, let us define a zipper that focuses on x and contains y everywhere else in its context:

```
let point = fun x y → cofunction : a zipper with
  | ..#Left → repeat y
  | ..#Proj → x
  | ..#Right → repeat y
```

Then, a specific instance of the Game of Life starting from a single alive cell and following our *rule* is defined as the infinite iteration over the uniform application obtained by the term *cobind rule*:

```
let rec iterate : type a.(a → a) → a → a stream = fun f a →
  cofunction : a stream with
  | ..#Head → a
  | ..#Tail → iterate f (f a)
let game_of_life = iterate (cobind rule) (point true false)
```

6 RELATED WORK

CoCaml [8] is an extension of the OCaml programming language with facilities to define functions over coinductive datatypes. We obviously share the same motivations as CoCaml: tackling the lack of support for infinite structures in this language. However, our approaches differ significantly. CoCaml only deals with *regular* coinductive datatypes, that is the subset of infinite values representable using cyclic values. By restricting itself to regular coinductive datatypes, CoCaml can reuse the ability of OCaml to define cyclic values. These representations are more efficient than ours since they are defined eagerly, not on demand. Another effect of this restriction is the opportunity for CoCaml to introduce a new construction **let corec** [*solver*] $f\ x =$ **match** x **with**... which transforms the pattern matching inside f into a set of equations which are subsequently solved using a *solver* specified by the programmer. This approach offers stronger guarantees than ours since the different solvers can check for instance that the defined computations over regular coinductive datatypes are terminating or preserve the regularity of infinite values.

We implement the same syntax as the original copatterns paper[1] but, even if we share the same syntax for full copattern matching, the operational semantics are different because we consider branches as a sequence, not as a set of equations. The system [2] introduces a notion of *generalized abstraction* which is similar to the functions and corecursive functions of our source language λ^{C}. The simple copatterns of λ^{C} are directly imported from the paper of Setzer et al. [14] that presents a program transformation to unnest deep copatterns into simple copatterns. As mentionned in Section 4.1, we do not implement this transformation as is because we want to stick with the ML pattern matching convention. From the typing perspective, our system is closer to the original system of indexed codata types[15].

To our knowledge, none of the existing languages with copatterns provides first-class first-order observation queries like ours and none of them studies the encoding of copattern matching in terms of pattern matching. Notice that our metatheoretical study is simpler than previous work about copatterns. Since we are extending OCaml and not a proof assistant like Coq [4] or Agda [12], we are looking for type safety and the correctness of our translation, rather than strong normalization or productivity.

7 CONCLUSION

Let us recall our contributions: this paper introduces an extension of OCaml that features copattern matching first-class observation requests. We have shown that this extension can be implemented by a local syntactic translation and that GADTs and second-order polymorphism are necessary to typecheck the programs generated by the translation. To stay close to the ML programming language, we have made design choices about unnesting and overlapping definitions which differ from previous work.

Even though our "macro" gives sufficient mechanisms to write observation-centric programs, this implementation choice forces the programmer to write type annotations that could be inferred by the type inference engine. As future work, we plan to move the translation a step deeper inside the compiler to be able to use contextual type information to automatically build type annotations on *dispatch* when possible.

Acknowledgement. We are grateful to the anonymous reviewers of a previous version of this paper for their great job in helping us improve our presentation. We also thank Gabriel Scherer for his enlightening feedback about this work.

REFERENCES

[1] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: programming infinite structures by observations. In *42nd ACM SIGPLAN conference on Principle of Programming Languages*, Vol. 48. ACM, 27–38.
[2] Andreas M Abel and Brigitte Pientka. 2013. Wellfounded recursion with copatterns: A unified approach to termination and productivity. In *18th ACM SIGPLAN International Conference on Functional Programming*, Vol. 48. ACM, 185–196.
[3] John Conway. 1970. The game of life. *Scientific American* 223, 4 (1970), 4.
[4] The Coq development team. 2009. *The Coq proof assistant reference manual*. LogiCal Project. http://coq.inria.fr/doc/
[5] Conal Eliott. 2008. Sequences, streams, and segments. (2008). http://conal.net/blog/posts/sequences-streams-and-segments
[6] Jacques Garrigue and Didier Rémy. 1999. Semi-explicit first-class polymorphism for ML. *Information and Computation* 155, 1-2 (1999), 134–169.
[7] Gérard Huet. 1997. The zipper. *Journal of functional programming* 7, 05 (1997), 549–554.
[8] Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. 2017. CoCaml: Functional Programming with Regular Coinductive Types. *Fundamenta Informaticae* 150 (2017), 347–377.
[9] Marina Lenisa, John Power, and Hiroshi Watanabe. 2000. Distributivity for endofunctors, pointed and co-pointed endofunctors, monads and comonads. *Electronic Notes in Theoretical Computer Science* 33 (2000), 230–260.
[10] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2014. The OCaml system release 4.02: Documentation and user's manual. (2014).
[11] Daniel R Licata, Noam Zeilberger, and Robert Harper. 2008. Focusing on binding and computation. In *Logic in Computer Science, 2008. LICS'08. 23rd Annual IEEE Symposium on*. IEEE, 241–252.
[12] Ulf Norell. 2009. Dependently Typed Programming in Agda. In *4th International Workshop on Types in Language Design and Implementation (TLDI '09)*. ACM, New York, NY, USA, 1–2.
[13] Dan Piponi. 2006. Evaluating cellular automata is comonadic. *Blog post* (2006). http://blog.sigfpe.com/2006/12/evaluating-cellular-automata-is.html
[14] Anton Setzer, Andreas Abel, Brigitte Pientka, and David Thibodeau. 2014. Unnesting of copatterns. In *International Conference on Rewriting Techniques and Applications*. Springer, 31–45.
[15] David Thibodeau, Andrew Cave, and Brigitte Pientka. 2016. Indexed codata types. In *21st ACM SIGPLAN International Conference on Functional Programming*. ACM, 351–363.
[16] Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded recursive datatype constructors. In *32nd ACM SIGPLAN conference on Principle of Programming Languages*, Vol. 38. ACM, 224–235.
[17] Noam Zeilberger. 2008. Focusing and higher-order abstract syntax. In *35th ACM SIGPLAN conference on Principle of Programming Languages*, Vol. 43. ACM, 359–369.