Friends with Benefits
Implementing Corecursion in Foundational Proof Assistants

Jasmin Christian Blanchette[1,2], Aymeric Bouzy[3], Andreas Lochbihler[4], Andrei Popescu[5,6], and Dmitriy Traytel[4]

[1] Vrije Universiteit Amsterdam, the Netherlands
[2] Inria Nancy – Grand Est, Nancy, France
[3] Laboratoire d'informatique, École polytechnique, Palaiseau, France
[4] Institute of Information Security, Department of Computer Science, ETH Zürich, Switzerland
[5] Department of Computer Science, Middlesex University London, UK
[6] Institute of Mathematics Simion Stoilow of the Romanian Academy, Bucharest, Romania

Abstract. We introduce AmiCo, a tool that extends a proof assistant, Isabelle/HOL, with flexible function definitions well beyond primitive corecursion. All definitions are certified by the assistant's inference kernel to guard against inconsistencies. A central notion is that of *friends*: functions that preserve the productivity of their arguments and that are allowed in corecursive call contexts. As new friends are registered, corecursion benefits by becoming more expressive. We describe this process and its implementation in detail, from the user's specification to the synthesis of a higher-order definition to the registration of a friend. We show some substantial case studies where our approach makes a difference.

1 Introduction

Codatatypes and corecursion [21, 29, 66] are emerging as a major methodology for programming with infinite objects. Unlike in traditional lazy functional programming, codatatypes support *total* (*co*)*programming* [1, 8, 33, 74], where the defined functions have a simple set-theoretic semantics and productivity is guaranteed. The proof assistants Agda [18], Coq [12], and Matita [7] have been supporting this methodology for years.

By contrast, proof assistants based on higher-order logic (HOL), such as HOL4 [69], HOL Light [35], and Isabelle/HOL [60], have traditionally provided only datatypes. Isabelle/HOL is the first of these systems to also offer codatatypes. It took two years, and about 24 000 lines of Standard ML, to move from an understanding of the mathematics [17, 73] to an implementation that automates the process of checking high-level user specifications and producing the necessary corecursion and coinduction theorems [15].

There are important differences between Isabelle/HOL and type theory systems such as Coq in the way they handle corecursion. Consider the codatatype of streams given by

$$\texttt{codatatype}\ \alpha\ \mathsf{stream} = (\mathsf{shd}\colon \alpha) \lhd (\mathsf{stl}\colon \alpha\ \mathsf{stream})$$

where $\lhd$ (written infix) is the constructor, and shd and stl are the head and tail selectors, respectively. In Coq, a definition such as

```
corec natsFrom : nat → nat stream where
  natsFrom n = n ◁ natsFrom (n + 1)
```

which introduces the function $n \mapsto n \lhd n+1 \lhd n+2 \lhd \cdots$, is accepted after a syntactic check that detects the $\lhd$-guardedness of the corecursive call. In Isabelle, this check is replaced by a deeper analysis. The `primcorec` command [15] transforms a user specification into a *blueprint* object: the coalgebra $b = \lambda n.\ (n, n+1)$. Then $\mathsf{natsFrom}$ is defined as $\mathsf{corec}_{\mathsf{stream}}\ b$, where $\mathsf{corec}_{\mathsf{stream}}$ is the fixed primitive corecursive combinator for α stream. Finally, the user specification is derived as a theorem from the definition and the characteristic equation of the corecursor.

Unlike in type theories, where (co)datatypes and (co)recursion are built-in, the HOL philosophy is to reduce every new construction to the core logic. This usually requires a lot of implementation work but guarantees that definitions introduce no inconsistencies. Since codatatypes and corecursion are derived concepts, there is no a priori restriction on the expressiveness of user specifications other than expressiveness of HOL itself.

Consider a variant of $\mathsf{natsFrom}$, where the function $\mathsf{add1} : \mathsf{nat} \to \mathsf{nat\ stream} \to \mathsf{nat\ stream}$ adds 1 to each element of a stream:

```
corec natsFrom : nat → nat stream where
  natsFrom n = n ◁ add1 (natsFrom n)
```

Coq's syntactic check fails on add1. After all, add1 could explore the tail of its argument before it produces a constructor, hence blocking productivity and leading to underspecification or inconsistency.

Isabelle's bookkeeping allows for more nuances. Suppose add1 has been defined as

```
corec add1 : nat stream → nat stream where
  add1 ns = (shd ns + 1) ◁ add1 (stl ns)
```

When analyzing add1's specification, the `corec` command synthesizes its definition as a blueprint b. This definition can then be proved to be *friendly*, hence acceptable in corecursive call contexts when defining other functions. Functions with friendly definitions are called friendly, or *friends*. These functions preserve productivity by consuming at most one constructor when producing one.

Our previous work [16] presented the category theory underlying friends, based on more expressive blueprints than the one shown above for primitive corecursion. We now introduce a tool, AmiCo, that automates the process of applying and incrementally improving corecursion.

To demonstrate AmiCo's expressiveness and convenience, we used it to formalize eight case studies in Isabelle, featuring a variety of codatatypes and corecursion styles (Sect. 2). A few of these examples required ingenuity and suggest directions for future work. Most of the examples fall in the executable framework of Isabelle, which allows for code extraction to Haskell via Isabelle's code generator. One of them pushes the boundary of executability, integrating friends in the quantitative world of probabilities.

At the low level, the *corecursion state* summarizes what the system knows at a given point, including the set of available friends and a corecursor *up to* friends (Sect. 3). Polymorphism complicates the picture, because some friends may be available only for specific instances of a polymorphic codatatype. To each corecursor corresponds a coinduction principle up to friends and a uniqueness theorem that can be used to reason about corecursive functions. All of the constructions and theorems are derived from first principles, without requiring new axioms or extensions of the logic. This *foundational approach* prevents the introduction of inconsistencies, such as those that have affected the termination and productivity checkers of Agda and Coq [23, 71] in recent years.

The user interacts with our tool via the following commands to the proof assistant (Sect. 4). The `corec` command defines a function f by extracting a blueprint *b* from a user's specification, defining f using *b* and a corecursor, and deriving the original specification from the characteristic property of the corecursor. Moreover, `corec` supports mixed recursion–corecursion specifications, exploiting proof assistant infrastructure for terminating (well-founded) recursion. Semantic proof obligations, notably termination, are either discharged automatically or presented to the user. Specifying the `friend` option to `corec` additionally registers f as a friend, enriching the corecursor state. Another command, `friend_of_corec`, registers existing functions as friendly. Friendliness amounts to the relational parametricity [64, 75] of a selected part of the definition [16], which in this paper we call a *surface*. The tool synthesizes the surface, and the parametricity proof is again either discharged automatically or presented to the user.

AmiCo is a significant piece of engineering, at about 7 000 lines of Standard ML code (Sect. 5). It subsumes a crude prototype [16] based on a shell script and template files that automated the corecursor derivation but left the blueprint and surface synthesis problems to the user. Our tool is available as part of the official Isabelle2016-1 release. The formalized examples and case studies are provided in an archive [14].

The contributions of this paper are the following:

– We describe our tool's design, algorithms, and implementation as a foundational extension of Isabelle/HOL, taking the form of the `corec`, `friend_of_corec`, `corecursive` and `coinduction_upto` commands and the *corec_unique* proof method.
– We apply our tool to a wide range of case studies, most of which are either beyond the reach of competing systems or would require type annotations and additional proofs.
– We rigorously justify mixed recursion–corecursion, a practically important notion that has been little considered in the literature.

Although our tool works for Isabelle, the same methodology is immediately applicable to any prover in the HOL family (including HOL4, HOL Light, HOL Zero [6], and HOL-Omega [37]), whose users represent about half of the proof assistant community. Moreover, a similar methodology is in principle applicable to provers based on type theory, such as Agda, Coq, and Matita (Sect. 6).

Conventions We recall the syntax relevant for this paper, relying on the standard set-theoretic interpretation of HOL [30].

We fix infinite sets of type variables $\alpha, \beta, \ldots$ and term variables $x, y, \ldots$ and a higher-order signature, consisting of a set of type constructors including bool and the binary constructors for functions ($\to$), products ($\times$), and sums ($+$). Types σ, τ are defined using type variables and applying type constructors, normally written postfix. We write $\sigma' \leq \sigma$ if σ' can be obtained from σ by applying a type substitution. Isabelle/HOL supports Haskell-style type classes, with :: expressing class membership (e.g., int :: ring).

Moreover, we assume a set of polymorphic constants c, f, g, ... with declared types, including equality $= : \alpha \to \alpha \to \mathsf{bool}$, left and right product projections fst and snd, and left and right sum embeddings Inl and Inr. Terms t are built from constants c and variables x by means of typed λ-abstraction and application. We write $(\Gamma \vdash)\ t : \sigma$ to indicate that term t has type σ (in context Γ). Polymorphic constants and terms will be freely used in contexts that require a less general type, without indicating the actual instance.

2 Motivating Examples

We apply AmiCo to eight case studies to demonstrate its benefits—in particular, the flexibility that friends provide and reasoning by uniqueness (of solutions to corecursive equations). The first four examples demonstrate the flexibility that friends provide. The third one also features reasoning by uniqueness. The fourth example crucially relies on a form of nested corecursion where the operator under definition must be recognized as a friend. The fifth through seventh examples mix recursion with corecursion and discuss the associated proof techniques. The last example, about a probabilistic process calculus, takes our tool to its limits: We discuss how to support corecursion through monadic sequencing and mix unbounded recursion with corecursion. All eight formalizations are available online [14], together with our earlier stream examples [16].

Since all examples are taken from the literature, we focus on the formalization with AmiCo. No detailed understanding is needed to see that they fit within the friends framework. Background information can be found in the referenced works.

Remarkably, none of the eight examples work with Coq's or Matita's standard mechanisms. Sized types in Agda [4] can cope with the first six but fail on the last two: In one case a function must inspect an infinite list unboundedly deeply, and in the other case the codatatype cannot even be defined in Agda. The Dafny verifier, which also provides codatatypes [49], supports only the seventh case study.

2.1 Coinductive Languages

Rutten [67] views formal languages as infinite tries, i.e., prefix trees branching over the alphabet with boolean labels at the nodes indicating whether the path from the root denotes a word in the language. The type α lang features corecursion through the right-hand side of the function arrow ($\to$).

codatatype $\alpha\ \mathsf{lang} = \mathsf{Lang}\ (o : \mathsf{bool})\ (\delta : \alpha \to \alpha\ \mathsf{lang})$

Traytel [72] has formalized tries in Isabelle using a codatatype, defined regular operations on them as corecursive functions, and proved by coinduction that the defined operations form a Kleene algebra. Because Isabelle offered only primitive corecursion when this formalization was developed, the definition of concatenation, iteration, and shuffle product was tedious, spanning more than a hundred lines.

Corecursion up to friends eliminates this tedium. The following extract from an Isabelle formalization is all that is needed to define the main operations on languages:

corec (friend) $+ : \alpha\ \mathsf{lang} \to \alpha\ \mathsf{lang} \to \alpha\ \mathsf{lang}$ where
  $L + K = \mathsf{Lang}\ (o\ L \vee o\ K)\ (\lambda a.\ \delta\ L\ a + \delta\ K\ a)$

corec (friend) $\cdot : \alpha\ \mathsf{lang} \to \alpha\ \mathsf{lang} \to \alpha\ \mathsf{lang}$ where

$L \cdot K = \mathsf{Lang}\ (o\ L \wedge o\ K)\ (\lambda a.\ \text{if } o\ L \text{ then } (\delta\ L\ a \cdot K) + \delta\ K\ a \text{ else } \delta\ L\ a \cdot K)$

`corec` `(friend)` $^* : \alpha\ \mathsf{lang} \to \alpha\ \mathsf{lang}$ `where`
  $L^* = \mathsf{Lang\ True}\ (\lambda a.\ \delta\ L\ a \cdot L^*)$

`corec` `(friend)` $\| : \alpha\ \mathsf{lang} \to \alpha\ \mathsf{lang} \to \alpha\ \mathsf{lang}$ `where`
  $L \parallel K = \mathsf{Lang}\ (o\ L \wedge o\ K)\ (\lambda a.\ (\delta\ L\ a \cdot K) + (L \cdot \delta\ K\ a))$

Concatenation $(\cdot)$ and shuffle product $(\|)$ are corecursive up to alternation $(+)$, and iteration $(^*)$ is corecursive up to concatenation $(\cdot)$. All four definitions use an alternative λ-based syntax for performing corecursion under the right-hand side of $\to$, instead of applying the functorial action $\mathsf{map}_{\to} = \circ$ (composition) associated with $\to$.

The `corec` command is provided by AmiCo, whereas `codatatype` and `primcorec` (Sect. 3.2) has been part of Isabelle since 2013. The `friend` option registers the defined functions as friends and automatically discharges the emerging proof obligations, which ensure that friends consume at most one constructor to produce one constructor.

Proving equalities on tries conveniently works by coinduction up to congruence (Sect. 3.7). Already before `corec`'s existence, Traytel was able to write automatic one-line proofs such as

`lemma` $K \cdot (L + M) = K \cdot L + K \cdot M$
  `by` (*coinduction arbitrary*: $K\ L\ M$ *rule*: $+$.*coinduct*) *auto*

The *coinduction* proof method [15] instantiates the bisimulation witness of the given coinduction rule before applying it backwards. Without `corec`, the rule $+$.*coinduct* of coinduction up to congruence had to be stated and proved manually, including the manual inductive definition of the congruence closure under $+$.

Overall, the usage of `corec` compressed Traytel's development from 750 to 600 lines of Isabelle text. In Agda, Abel [3] has formalized Traytel's work up to proving the recursion equation $L^* = \varepsilon + L \cdot L^*$ for iteration $(^*)$ in 219 lines of Agda text, which correspond to 125 lines in our version. His definitions are as concise as ours, but his proofs require more manual steps.

2.2 Knuth–Morris–Pratt String Matching

Building on the trie view of formal languages, van Laarhoven [47] discovered a concise formulation of the Knuth–Morris–Pratt algorithm [44] for finding one string in another:

$\mathsf{is\_substring\_of}\ xs\ ys = \mathsf{match}\ (\mathsf{mk\_table}\ xs)\ ys$

$\mathsf{match}\ t\ xs = (o\ t \vee (xs \neq [] \wedge \mathsf{match}\ (\delta\ t\ (\mathsf{hd}\ x))\ (\mathsf{tl}\ xs)))$

$\mathsf{mk\_table}\ xs = \text{let } \mathsf{table} = \mathsf{tab}\ xs\ (\lambda\_.\ \mathsf{table}) \text{ in } \mathsf{table}$

$\mathsf{tab}\ []\ f \qquad\quad = \mathsf{Lang\ True}\ f$
$\mathsf{tab}\ (x \lhd xs)\ f = \mathsf{Lang\ False}\ (\lambda c.\ \text{if } c = x \text{ then } \mathsf{tab}\ xs\ (\delta\ (f\ x)) \text{ else } f\ c)$

Here, we overload the stream constructor $\lhd$ for finite lists; hd and tl are the selectors. In our context, $\mathsf{table} : \alpha\ \mathsf{lang}$ is the most interesting definition because it corecurses through tab. Since there is no constructor guard, table would appear not to be productive. However, the constructor is merely hidden in tab and can be pulled out by unrolling the definition of tab as follows.

As the first step, we register Δ defined by $\Delta\ xs\ f = \delta\ (\mathsf{tab}\ xs\ f)$ as a friend, using the `friend_of_corec` command provided by our tool. The registration of an existing function as a friend requires us to supply an equation with a constructor-guarded right-hand side and to prove the equation and the parametricity of the destructor-free part of the right-hand side, called the surface (Sect. 3.4). Then the definition of table corecurses through Δ. Finally, we derive the original specification by unrolling the definition. We can use the derived specification in the proofs, because proofs in HOL do not depend on the actual definition (unlike in type theory).

```
corec tab : α list → (α → α lang) → α lang where
  tab xs f = Lang (xs = []) (λc. if xs = [] ∨ hd xs ≠ c then f c else tab (tl xs) (δ (f c)))

definition Δ : α list → (α → α lang) → α → α lang where
  Δ xs f = δ (tab xs f)

friend_of_corec Δ where
  Δ xs f c = Lang
    (if xs = [] ∨ hd xs ≠ c then o (f x) else tl xs = [])
    (if xs = [] ∨ hd xs ≠ c then δ (f x) else Δ (tl xs) (δ (f c)))
  ⟨two-line proof of the equation and of parametricity⟩

context fixes xs : α list begin
  corec table : α lang where
    table = Lang (xs = []) (Δ xs (λ_. table))

  lemma table = tab xs (λ_. table)
    ⟨one-line proof⟩
end
```

2.3 The Stern–Brocot Tree

The next application involves infinite trees of rational numbers. It is based on Hinze's work on the Stern–Brocot and Bird trees [36] and the Isabelle formalization by Gammie and Lochbihler [27]. It illustrates reasoning by uniqueness (Sect. 3.7).

The Stern–Brocot tree contains all the rational numbers in their lowest terms. It is an infinite binary tree frac tree of formal fractions frac = nat × nat. Each node is labeled with the mediant of its rightmost and leftmost ancestors, where mediant $(a, c)\ (b, d) = (a + b, c + d)$. Gammie and Lochbihler define the tree via an iterative helper function.

```
codatatype α tree = Node (root: α) (left: α tree) (right: α tree)

primcorec stern_brocot_gen : frac → frac → frac tree where
  stern_brocot_gen l u =
    let m = mediant l u in Node m (stern_brocot_gen l m) (stern_brocot_gen m u)

definition stern_brocot : frac tree where
  stern_brocot = stern_brocot_gen (0, 1) (1, 0)
```

Using AmiCo, we can directly formalize Hinze's corecursive specification of the tree, where nxt $(m, n) = (m + n, n)$ and swap $(m, n) = (n, m)$. The tree is corecursive up to the two friends suc and $1\,/\,t$.

```
corec (friend) suc : frac tree → frac tree where
  suc t = Node (nxt (root t)) (suc (left t)) (suc (right t))
```

```
corec (friend) 1 / _ : frac tree → frac tree where
  1 / t = Node (swap (root t)) (1 / left t) (1 / right t)
```

```
corec stern_brocot : frac tree where
  stern_brocot = Node (1, 1) (1 / (suc (1 / stern_brocot))) (suc stern_brocot)
```

Without the iterative detour, the proofs, too, become more direct as the statements need not be generalized for the iterative helper function. For example, Hinze relies on the uniqueness principle to show that a loopless linearization stream stern_brocot of the tree yields Dijkstra's fusc function [25] given by

$$\mathsf{fusc} = 1 \lhd \mathsf{fusc}' \qquad \mathsf{fusc}' = 1 \lhd (\mathsf{fusc} + \mathsf{fusc}' - 2 \cdot (\mathsf{fusc} \bmod \mathsf{fusc}'))$$

where all arithmetic operations are lifted to streams elementwise—e.g., $xs + ys = \mathsf{map}_{\mathsf{stream}}\ (+)\ (xs \between ys)$, where $\between$ zips two streams. We define fusc and stream as follows. To avoid the mutual corecursion, we inline fusc in fusc$'$ for the definition with `corec`, after having registered the arithmetic operations as friends:

```
corec fusc′ : nat stream where
  fusc′ = 1 ◁ ((1 ◁ fusc′) + fusc′ − 2 · ((1 ◁ fusc′) mod fusc′))
```

```
definition fusc : nat stream where
  fusc = 1 ◁ fusc′
```

```
corec chop : α tree → α tree where
  chop (Node x l r) = Node (root l) r (chop l)
```

```
corec stream : α tree → α stream where
  stream t = root t ◁ stream (chop t)
```

Hinze proves that stream stern_brocot equals fusc $\between$ fusc$'$ by showing that both satisfy the corecursion equation $x = (1, 1) \lhd \mathsf{map}_{\mathsf{stream}}\ \mathsf{step}\ x$, where $\mathsf{step}\ (m, n) = (n, m + n - 2 \cdot (m \bmod n))$. This equation yields the loopless algorithm, because siterate step $(1, 1)$ satisfies it as well, where siterate is defined by

```
primcorec siterate : (α → α) → α → α stream where
  siterate f x = x ◁ siterate f (f x)
```

Our tool generates a proof rule for uniqueness of solutions to the recursion equation (Sect. 3.7). We conduct the equivalence proofs using this rule.

For another example, all rational numbers also occur in the Bird tree given by

```
corec bird : frac tree where
  bird = Node (1, 1) (1 / suc bird) (suc (1 / bird))
```

It satisfies 1 / bird = mirror bird, where mirror corecursively swaps all all left and right subtrees of a tree. Again, we prove this identity by showing that both sides satisfy the corecursion equation $x = \mathsf{Node}\ (1, 1)\ (\mathsf{suc}\ (1 / x))\ (1 / \mathsf{suc}\ x)$. This equation does not correspond to any function defined with `corec`, but we can derive its uniqueness principle using our proof method *corec_unique* without defining the function. The resulting Isabelle proof is quite concise:

```
let ?H = λx. Node (1, 1) (suc (1 / x)) (1 / suc x)
have mb: mirror bird = ?H (mirror bird)          by (rule tree.expand) ...
have unique: ∀t. t = ?H t ⟶ t = mirror bird    by corec_unique (fact mb)
have 1 / bird = ?H (1 / bird)                    by (rule tree.expand) ...
then show 1 / bird = mirror bird                 by (rule unique)
```

No coinduction is needed: The identities are proved by expanding the definitions a finite number of times (once each here). Following Hinze's proof, we also show that odd_mirror bird = stern_brocot by uniqueness, where odd_mirror swaps the subtrees only at levels of odd depth.

Gammie and Lochbihler manually derive each uniqueness rule using a separate coinduction proof. For odd_mirror alone, the proof requires 25 lines. With AmiCo's *corec_unique* proof method, such proofs are automatic.

2.4 Breadth-First Tree Labeling

Abel and Pientka [4] demonstrate the expressive power of sized types in Agda with the example of labeling the nodes of an infinite binary tree in breadth-first order, which they adapted from Jones and Gibbons [42]. The function bfs takes a stream of streams of labels as input and labels the nodes at depth *i* according to a prefix of the *i*th input stream. It also outputs the streams of unused labels. Then bf ties the knot by feeding the unused labels back into bfs:

$$\mathsf{bfs}\ ((x \lhd xs) \lhd ys) = \\ \quad \text{let}\ (l, ys') = \mathsf{bfs}\ ys;\ (r, ys'') = \mathsf{bfs}\ ys'\ \text{in}\ (\mathsf{Node}\ x\ l\ r,\ xs \lhd ys'')$$

$$\mathsf{bf}\ xs = \text{let}\ (t, lbls) = \mathsf{bfs}\ (xs \lhd lbls)\ \text{in}\ t$$

Because bfs returns a pair, we define the two projections separately and derive the original specification for bfs trivially from the definitions. One of the corecursive calls to bfs_2 occurs in the context of bfs_2 itself—it is "self-friendly" (Sect. 4.2).

```
corec (friend) bfs₂ : α stream stream → α stream stream
  where bfs₂ ((x ◁ xs) ◁ ys) = xs ◁ bfs₂ (bfs₂ ys)

corec bfs₁ : α stream stream → α tree where
  bfs₁ ((x ◁ xs) ◁ ys) = Node x (bfs₁ ys) (bfs₁ (bfs₂ ys))

definition bfs : α stream → α tree where
  bfs xss = (bfs₁ xss, bfs₂ xss)

corec labels : α stream → α stream stream where
  labels xs = bfs₂ (xs ◁ labels xs)

definition bf : α stream → α tree where
  bf xs = bfs₁ (xs ◁ labels xs)
```

For comparison, Abel's and Pientka's formalization in Agda is of similar size, but the user must provide some size hints for the corecursive calls.

2.5 Stream Processors

Stream processors are a standard example of mixed fixpoints:

```
datatype $(\alpha, \beta, \delta)$ sp$_\mu$ = Get $(\alpha \to (\alpha, \beta, \delta)$ sp$_\mu)$ | Put $\beta\ \delta$
codatatype $(\alpha, \beta)$ sp$_\nu$ = In (out: $(\alpha, \beta, (\alpha, \beta)$ sp$_\nu)$ sp$_\mu$)
```

When defining functions on these objects, we previously had to break them into a recursive and a corecursive part, using Isabelle's `primcorec` command for the latter [15]. Since our tool supports mixed recursion–corecursion, we can now express functions on stream processors more directly.

We present two functions. The first one runs a stream processor:

```
corecursive run : $(\alpha, \beta)$ sp$_\nu \to \alpha$ stream $\to \beta$ stream where
  run $sp\ s$ = case out $sp$ of
      Get $f \Rightarrow$ run (In ($f$ (shd $s$))) (stl $s$)
    | Put $b\ sp \Rightarrow b \lhd$ run $sp\ s$
  ⟨two-line termination proof⟩
```

The second function, $\circ\circ$, composes two stream processors:

```
corec (friend) get where
  get $f$ = In (Get ($\lambda a$. out ($f\ a$)))

corecursive $\circ\circ$ : $(\beta, \gamma)$ sp$_\nu \to (\alpha, \beta)$ sp$_\nu \to (\alpha, \gamma)$ sp$_\nu$ where
  $sp \circ\circ sp'$ = case (out $sp$, out $sp'$) of
      (Put $b\ sp$, _) $\Rightarrow$ In (Put $b$ ($sp \circ\circ sp'$))
    | (Get $f$, Put $b\ sp'$) $\Rightarrow$ In ($f\ b$) $\circ\circ\ sp'$
    | (_, Get $f'$) $\Rightarrow$ get ($\lambda a$. $sp \circ\circ$ In ($f'\ a$))
  ⟨two-line termination proof⟩
```

The selector out in the noncorecursive friend get is legal, because get also adds a constructor. In both cases, the `corecursive` command emits a termination proof obligation, which we discharged in two lines, using the same techniques as when defining recursive functions. This command is equivalent to `corec`, except that it lets the user discharge proof obligations instead of applying some standard proof automation.

2.6 A Calculator

Next, we formalize a calculator example by Hur et al. [40]. The calculator inputs a number, computes the double of the sum of all inputs, and outputs the current value of the sum. When the input is 0, the calculator counts down to 0 and starts again. Hur et al. implement two versions, f and g, in a programming language embedded deeply in Coq and prove that f simulates g using parameterized coinduction.

We model the calculator in a shallow fashion as a function from the current sum to a stream processor for nats. Let calc abbreviate nat $\to$ (nat, nat) sp$_\nu$. We can write the program directly as a function and very closely to its specification [40, Figure 2]. In f and g, the corecursion goes through the friends get and restart, and the constructor guard is hidden in the abbreviation put $x\ sp$ = In (Put $x\ sp$).

```
corec (friend) restart : calc → calc where
  restart $h\ n$ = if $n > 0$ then put $n$ (restart $h$ $(n-1)$) else $h$ 0
corec f : calc where
  f $n$ = put $n$ (get ($\lambda v$. if $v \neq 0$ then f $(2 \cdot v + n)$ else restart f $(v + n)$))
corec g : calc where
  g $m$ = put $(2 \cdot m)$ (get ($\lambda v$. if $v = 0$ then restart g $(2 \cdot m)$ else g $(v + m)$))
```

Our task is to prove that g m simulates f $(2 \cdot m)$. In fact, the two can even be proved to be bisimilar. In our shallow embedding, bisimilarity coincides with equality. We prove g m = f $(2 \cdot m)$ by coinduction with the rule generated for the friends get and restart. The proof exploits the fact that restart applies its argument only to 0.

```
lemma restart_cong: $h\ 0 = k\ 0 \longrightarrow$ restart $h\ n$ = restart $k\ n$
  by (induction n) (simp_all add: restart.code)
lemma g $m$ = f $(2 \cdot m)$
proof (coinduction arbitrary: m rule: restart.coinduct)
  case (Eq_nat_nat_sp$_v$ m)
  have restart g $(2 \cdot m)$ = restart ($\lambda$_. g 0) $(2 \cdot m)$ and
       restart f $(2 \cdot m)$ = restart ($\lambda$_. f 0) $(2 \cdot m)$
    by (rule restart_cong; simp)+
  then show ?case by (auto intro!: sp$_v$.cong_restart ...)
qed
```

2.7 Lazy List Filtering

A classic example requiring a mix of recursion and corecursion is filtering on lazy lists. Given the polymorphic type of lazy lists

```
codatatype $\alpha$ llist = [] | (lhd: $\alpha$) ◁ (ltl: $\alpha$ llist)
```

the task is to define the function lfilter : $(\alpha \to \text{bool}) \to \alpha\ \text{llist} \to \alpha\ \text{llist}$ that retains only the elements that satisfy the given predicate. Devillers et al. [24] noted the difficulty of defining lfilter and proving its properties. Paulson [62] defined lfilter using an inductive search predicate. His development culminates in a proof of

$$\text{lfilter}\ P \circ \text{lfilter}\ Q = \text{lfilter}\ (\lambda x.\ P\ x \wedge Q\ x) \qquad (1)$$

Matthews [54] defines lfilter using contractions for converging equivalence relations. His contraction proof relies on a search function that returns the first index of an element satisfying the predicate. In Dafny, Leino [48] suggests a definition that mixes recursion and corecursion; his termination proof uses a search function similar to Matthews's. We can easily replicate Leino's definition in Isabelle, where set converts lazy lists to sets:

```
corecursive lfilter : $(\alpha \to$ bool$) \to \alpha$ llist $\to \alpha$ llist where
  lfilter $P\ xs$ = if $\forall x \in$ set $xs$. $\neg\ P\ x$ then []
                  else if $P$ (lhd $xs$) then lhd $xs$ ◁ lfilter $P$ (ltl $xs$)
                  else lfilter $P$ (ltl $xs$)
  ⟨13-line termination proof⟩
```

The nonexecutability of the infinite $\forall$ quantifier in the 'if' condition is unproblematic in HOL, which has no built-in notion of computation; Agda, Coq, and Matita would reject the definition outright. Lochbihler and Hölzl [51] define $\mathsf{lfilter}$ as a least fixpoint in the prefix order on llist. Using five properties, they substantiate that fixpoint induction leads to shorter proofs than Paulson's approach.

We show how to prove three of their properties using our definition, namely (1) and

$$\mathsf{lfilter}\ P\ xs = [] \longleftrightarrow (\forall x \in \mathsf{set}\ xs.\ \neg\ P\ xs) \tag{2}$$

$$\mathsf{set}\ (\mathsf{lfilter}\ P\ xs) = \mathsf{set}\ xs \cap \{x \mid P\ x\} \tag{3}$$

We start with (2). We prove the interesting direction, $\longrightarrow$, by induction on $x \in \mathsf{set}\ xs$, where the inductive cases are solved automatically. For (3), the $\supseteq$ direction is also a simple induction on set. The other direction requires two nested inductions: first on $x \in \mathsf{set}\ (\mathsf{lfilter}\ P\ xs)$ and then a well-founded induction on the termination argument for the recursion in $\mathsf{lfilter}$. Finally, we prove (1) using the uniqueness principle. We first derive the uniqueness rule for $\mathsf{lfilter}$ by a coinduction with a nested induction; this approach reflects the mixed recursive-corecursive definition of $\mathsf{lfilter}$, which nests recursion inside corecursion.

```
lemma lfilter_unique:
  (∀xs. f xs = if ∀x ∈ set xs. ¬ P x then []
               else if P (lhd xs) then lhd xs ◁ f (ltl xs)
               else f (ltl xs)) ⟶
  f = lfilter P
```

(Our tool does not yet generate uniqueness rules for mixed recursive–corecursive definitions.) Then the proof of (1) is automatic:

```
lemma lfilter P ∘ lfilter Q = lfilter (λx. P x ∧ Q x)
  by (rule lfilter_unique) (auto elim: llist.set_cases)
```

Alternatively, we could have proved (1) by coinduction with a nested induction on the termination argument. The uniqueness principle works well because it incorporates both the coinduction and the induction. This underlines that uniqueness can be an elegant proof principle for mixed recursive–corecursive definitions, despite being much weaker than coinduction in the purely corecursive case. Compared with Lochbihler and Hölzl's proofs by fixpoint induction, our proofs are roughly of the same length, but `corecursive` eliminates the need for the lengthy setup for the domain theory.

2.8 Generative Probabilistic Values

Our final example relies on a codatatype that fully exploits Isabelle's modular datatype architecture built on bounded natural functors (Sect. 3.1) and that cannot be defined easily, if at all, in other systems.

Lochbihler [50] proposes generative probabilistic values (GPVs) as a semantic domain for probabilistic input–output systems. He defines GPVs coinductively as follows:

```
codatatype $(\alpha, \gamma, \rho)$ gpv $=$ GPV (gpv: $(\alpha + \gamma \times (\alpha, \gamma, \rho)$ rpv) spmf)
type_synonym $(\alpha, \gamma, \rho)$ rpv $= \rho \to (\alpha, \gamma, \rho)$ gpv
```

The type α spmf, of discrete subprobability distributions over α, is both a bounded natural functor and a monad. Conceptually, each GPV chooses probabilistically between failing, terminating with a result of type α, and continuing by producing an output γ and transitioning into a reactive probabilistic value (RPV), which waits for a response ρ of the environment before moving to the generative successor state.

Next, Lochbihler defines a monadic language for modeling cryptographic games as GPVs similar to a probabilistic coroutine monad. Nondeterministic environments can be refined to stateful probabilistic ones of type $(\sigma, \gamma, \rho)\ \mathsf{env} = \sigma \to \gamma \to (\rho \times \sigma)\ \mathsf{spmf}$, which represent cryptographic oracles; for each input γ, the oracle probabilistically produces a response ρ and updates its local state σ. He also considers stateful environment converters of type $(\sigma, \gamma, \rho, \gamma', \rho')\ \mathsf{conv} = \sigma \to \gamma \to (\rho \times \sigma, \gamma', \rho')\ \mathsf{gpv}$. They intercept outputs γ of and produce responses ρ for a GPV. In doing so, they themselves interact with an environment through outputs γ' and responses ρ'. The converter maintains a local state σ that persists between intercepts.

Lochbihler defines two composition operators for converters: Composition with an oracle yields a new oracle, and inlining into a GPV yields a new GPV. We focus on inlining, because oracle composition can be defined in terms of it. The operator $\mathsf{inline} : (\sigma, \gamma, \rho, \gamma', \rho')\ \mathsf{conv} \to (\alpha, \gamma, \rho)\ \mathsf{gpv} \to \sigma \to (\alpha \times \sigma, \gamma', \rho')\ \mathsf{gpv}$ takes a converter T, a GPV, and the initial state of the converter and returns a GPV, which produces a result α and T's new state after interacting with T's environment. Its corecursive definition poses two challenges: First, a corecursive call appears under the monadic sequencing operation $\gg\!\!=_{\mathsf{gpv}}$ on GPV. Second, it recursively searches the interactions between the GPV and T, i.e., mixes recursion and corecursion.

For the former, we cannot register $\gg\!\!=_{\mathsf{gpv}}$ as a friend directly, because its type, $(\alpha, \gamma, \rho)\ \mathsf{gpv} \to (\alpha \to (\beta, \gamma, \rho)\ \mathsf{gpv}) \to (\beta, \gamma, \rho)\ \mathsf{gpv}$, contains a type variable, α, that does not occur in the result codatatype (β, γ, ρ) gpv. AmiCo's implementation cannot handle additional type variables, because all type variables in the right-hand sides of HOL constant definitions must occur in the signature of the constant; thus, the corecursor and the congruence closure would need to take additional type tokens as arguments. Instead, we define a copy of gpv with a phantom type parameter β and the conversion functions gpv and gpv$'$. The copy $\gg\!\!=_{\mathsf{gpv}'}$ of $\gg\!\!=_{\mathsf{gpv}}$ can be registered as a friend using AmiCo's friend_of_corec command:

```
codatatype $(\alpha, \gamma, \rho, \beta)$ gpv$'$ $=$ GPV$'$ (the$'$: $(\alpha + \gamma \times (\rho \to (\alpha, \gamma, \rho, \beta)$ gpv$'))$ spmf)
definition $\gg\!\!=_{\mathsf{gpv}'} : (\alpha, \gamma, \rho)\ \mathsf{gpv} \to (\alpha \to (\beta, \gamma, \rho, \alpha)\ \mathsf{gpv}') \to (\beta, \gamma, \rho, \alpha)\ \mathsf{gpv}'$
  where $g \gg\!\!=_{\mathsf{gpv}'} f = \mathsf{gpv}'\ (g \gg\!\!=_{\mathsf{gpv}} (\mathsf{gpv} \circ f))$
friend_of_corec $\gg\!\!=_{\mathsf{gpv}'}$ where
  $g \gg\!\!=_{\mathsf{gpv}'} f = \mathsf{GPV}'\ (\mathsf{map_{spmf}}\ (\mathsf{map_+}\ \mathsf{id}\ (\mathsf{map_\times}\ \mathsf{id}$
      $(\lambda r\ x.\ \text{case}\ r\ x\ \text{of}\ \mathsf{Inl}\ g' \Rightarrow g' \mid \mathsf{Inr}\ g' \Rightarrow g' \gg\!\!=_{\mathsf{gpv}'} f)))$
    $(\mathsf{gpv}\ g \gg\!\!=_{\mathsf{spmf}} (\lambda x.\ \text{case}\ x\ \text{of}$
        $\mathsf{Pure}\ a \Rightarrow \mathsf{map_{spmf}}\ (\mathsf{map_+}\ \mathsf{id}\ (\mathsf{map_\times}\ \mathsf{id}\ (\mathsf{Inl} \circ)))\ (\mathsf{the}'\ (f\ a))$
      $\mid \mathsf{IO}\ o\ c \Rightarrow \mathsf{return_{spmf}}\ (\mathsf{IO}\ o\ (\mathsf{Inr} \circ c)))))$
```

where $\mathsf{Pure} = \mathsf{Inl}$ and $\mathsf{IO}\ o\ r = \mathsf{Inr}\ (o, r)$. The equation specified in `friend_of_corec` reformulates the definition of $\ggg\!=_{\mathsf{gpv}'}$ to have the expected format, with a GPV' guard outside. A proof obligation connects the two specifications. We define inlining on the copy with our tool (see below) and transport it back to gpv via

```
definition $\mathsf{inline}\ T\ gpv\ s = \mathsf{gpv}\ (\mathsf{inline}'\ T\ (\mathsf{gpv}'\ gpv)\ s)$
```

Inlining combines recursion and corecursion as follows: The recursive part goes through the interactions between the GPV and the converter and searches for the next interaction between the converter and its environment. The corecursive part performs the interaction found and iterates the search. Following Lochbihler, we separate the iteration from the search:

```
partial_function (spmf) search : $(\sigma, \gamma, \rho, \gamma', \rho')\ \mathsf{conv} \to (\alpha, \gamma, \rho)\ \mathsf{gpv} \to \sigma \to$
    $(\alpha \times \sigma + \gamma' \times (\rho \times \sigma, \gamma', \rho')\ \mathsf{rpv} \times (\alpha \to (\alpha, \gamma, \rho, \rho \times \sigma)\ \mathsf{gpv}')\ \mathsf{spmf}$ where
  $\mathsf{search}\ c\ v\ s = \mathrm{do}\ \{$
    $z \leftarrow \mathsf{gpv}\ v;$
    case $z$ of $\mathsf{Pure}\ x \Rightarrow \mathsf{return}_{\mathsf{spmf}}\ (\mathsf{Inl}\ (x, s))$
     $|\ \mathsf{IO}\ a\ r \Rightarrow \mathrm{do}\ \{$
       $y \leftarrow \mathsf{gpv}\ (c\ s\ a);$
       case $y$ of $\mathsf{Pure}\ (b, s') \Rightarrow \mathsf{search}\ c\ (r\ b)\ s'$
        $|\ \mathsf{IO}\ u\ r' \Rightarrow \mathsf{return}_{\mathsf{spmf}}\ (\mathsf{Inr}\ (u, r', \mathsf{gpv}' \circ r))\ \}\ \}$
```

```
corec $\mathsf{inline}' : (\sigma, \gamma, \rho, \gamma', \rho')\ \mathsf{conv} \to (\alpha, \gamma, \rho, \rho \times \sigma)\ \mathsf{gpv}' \to \sigma \to$
    $(\alpha \times \sigma, \gamma, \rho, \rho \times \sigma)\ \mathsf{gpv}'$ where
  $\mathsf{inline}'\ T\ g\ s = \mathsf{GPV}\ (\mathsf{map}_{\mathsf{spmf}}\ (\lambda x.\ \mathrm{case}\ x\ \mathrm{of}\ \mathsf{Inl}\ xs \Rightarrow \mathsf{Pure}\ xs$
    $|\ \mathsf{Inr}\ (o, T', r) \Rightarrow \mathsf{IO}\ o\ (\lambda x.\ T'\ x \ggg\!=_{\mathsf{gpv}'} (\lambda(x, s').\ \mathsf{inline}'\ T\ (r\ x)\ s')))\ (\mathsf{search}\ T\ g\ s)$
```

First, we formalize iteration by the corecursive function inline', in which the corecursive call occurs in the second argument of the friend $\ggg\!=_{\mathsf{gpv}'}$. Second, the search is captured by the recursive function search. If the GPV terminates with result x, there are no calls and the search terminates; otherwise, the GPV outputs a and becomes the RPV r. In that case, search analyzes T under the argument a. If T returns b without issuing an output itself, the search continues recursively on $r\ b$. Otherwise, the first output is found and the search terminates. The monadic definition of search relies on Isabelle's `partial_function` command [46], which internally uses the least fixpoint operator in the chain-complete partial order of discrete subprobability distributions.

For well-founded recursion, AmiCo automatically separates the recursive parts from the corecursive parts in a specification given to `corec` (Sect. 4.1). As the recursion in search is not well-founded, we perform this separation manually.

Inlining is associative: $\mathsf{inline}\ c_1\ (\mathsf{inline}\ c_2\ v\ s_2)\ s_1 = \mathsf{inline}\ (\mathsf{inline}\ c_1 \circ\!\circ c_2)\ v\ (s_2, s_1)$. (We omit reassociations of tuples for clarity; $f \circ\!\circ g$ denotes $\lambda(x, y)\ z.\ f\ (g\ x\ z)\ y$.) We prove this identity on the copy gpv', i.e., with inline' instead of inline, by coinduction up to congruence. The corecursive step reuses Lochbihler's lemmas about search, which are proved by fixpoint induction. The mix of least fixpoint and corecursion does not admit a strong uniqueness rule for showing the identity, because the least fixpoint is not the only solution to the equation of search.

Initially, Lochbihler had manually derived the coinduction rule up to $\ggg=_{\mathsf{gpv}}$, which our tool now generates. However, because of the copied type, our reformulation ended up roughly as complicated as the original. Moreover, we noted that coinduction up to congruence works only for equality; for user-defined predicates (e.g., typing judgments), the coinduction rule must still be derived manually. But even though this case study is not conclusive, it demonstrates the flexibility of the framework.

3 The Low Level: Corecursor States

Starting from the primitive corecursor provided by Isabelle [15], our tool derives corecursors up to larger and larger sets of friends. The corecursion state includes the set of friends $\mathcal{F}$ and the corecursor $\mathsf{corec}_{\mathcal{F}}$. Four operations manipulate states:

– BASE gives the first nonprimitive corecursor by registering the first friends—the constructors (Sect. 3.3);
– STEP incorporates a new friend into the corecursor (Sect. 3.4);
– MERGE combines two existing sets of friends (Sect. 3.5);
– INSTANTIATE specializes the corecursor type (Sect. 3.6).

The operations BASE and STEP have already been described in detail and with many examples in our previous paper [16]. Here, we give a brief, self-contained account of them. MERGE and INSTANTIATE are new operations whose need became apparent in the course of implementation.

3.1 Bounded Natural Functors

The mathematics behind our tool assumes that the considered type constructors are both functors and relators, that they include basic functors such as identity, constant, sum, and product, and that they are closed under least and greatest fixpoints (initial algebras and final coalgebras). The tool satisfies this requirement by employing Isabelle's infrastructure for bounded natural functors (BNFs) [15, 73]. For example, the codatatype α stream is defined as the greatest solution to the fixpoint equation $\beta \cong \alpha \times \beta$, where both the right-hand side $\alpha \times \beta$ and the resulting type α stream are BNFs.

BNFs have both a functor and a relator structure. If K is a unary type constructor, we assume the existence of polymorphic constants for the functorial action, or map function, $\mathsf{map}_{\mathsf{K}} : (\alpha \to \beta) \to \alpha\ \mathsf{K} \to \beta\ \mathsf{K}$ and the relational action, or relator, $\mathsf{rel}_{\mathsf{K}} : (\alpha \to \beta \to \mathsf{bool}) \to \alpha\ \mathsf{K} \to \beta\ \mathsf{K} \to \mathsf{bool}$, and similarly for n-ary type constructors. For finite lists, $\mathsf{map}_{\mathsf{list}}$ is the familiar map function, and given a relation r, $\mathsf{rel}_{\mathsf{list}}\ r$ relates two lists of the same length and with r-related elements positionwise. While the BNFs are functors on their covariant positions, the relator structure covers contravariant positions as well.

We assume that some of the polymorphic constants are known to be (relationally) *parametric* in some type variables, in the standard sense [64]. For example, if K is a ternary relator and $\mathsf{c} : (\alpha, \beta, \gamma)\ \mathsf{K}$, then c is parametric in β if $\mathsf{rel}_{\mathsf{K}}\ (=)\ r\ (=)\ \mathsf{c}\ \mathsf{c}$ holds for all $r : \beta \to \beta' \to \mathsf{bool}$. We also write $\mathsf{rel}_{(\alpha,\_,\gamma)\ \mathsf{K}}\ r\ \mathsf{c}\ \mathsf{c}$ to mean the same thing. In a slight departure from standard practice, if a term does not depend on a type variable α, we consider it parametric in α. The map function of a BNF is parametric in all its type variables. By contrast, $= : \alpha \to \alpha \to \mathsf{bool}$ is not parametric in α.

3.2 Codatatypes and Primitive Corecursion

We fix a codatatype J. In general, J may depend on some type variables, but we leave this dependency implicit for now. While J also may have multiple, curried constructors, it is viewed at the low level as a codatatype with a single constructor $\mathsf{ctor_J} : \mathsf{J}\ \mathsf{K_{ctor}} \to \mathsf{J}$ and a destructor $\mathsf{dtor_J} : \mathsf{J} \to \mathsf{J}\ \mathsf{K_{ctor}}$:

$\mathtt{codatatype}\ \mathsf{J} = \mathsf{ctor_J}\ (\mathsf{dtor_J}\colon \mathsf{J}\ \mathsf{K_{ctor}})$

The mutually inverse constructor and destructor establish the isomorphism between J and $\mathsf{J}\ \mathsf{K_{ctor}}$. For streams, we have $\beta\ \mathsf{K_{ctor}} = \alpha \times \beta$, $\mathsf{ctor}\ (h, t) = h \lhd t$, and $\mathsf{dtor}\ xs = (\mathsf{shd}\ xs, \mathsf{stl}\ xs)$. Low-level constructors and destructors combine several high-level constructors and destructors in one constant each. Internally, the `codatatype` command works on the low level, providing the high-level constructors as syntactic sugar [15].

In addition, the `codatatype` command derives a primitive corecursor $\mathsf{corec_J} : (\alpha \to \alpha\ \mathsf{K_{ctor}}) \to \alpha \to \mathsf{J}$ characterized by the equation $\mathsf{corec_J}\ b = \mathsf{ctor} \circ \mathsf{map_{K_{ctor}}}\ (\mathsf{corec_J}\ b) \circ b$. The `primcorec` command, provided by Isabelle, reduces a primitively corecursive specification to a plain, acyclic definition expressed using this corecursor.

3.3 Corecursion up to Constructors

We call blueprints the arguments passed to corecursors. When defining a corecursive function f, a *blueprint* for f is produced, and f is defined as the corecursor applied to the blueprint. The expressiveness of a corecursor is indicated by the codomain of its blueprint argument. The blueprint passed to the primitive corecursor must return an $\alpha\ \mathsf{K_{ctor}}$ value—e.g., a pair $(m, x) : \mathsf{nat} \times \alpha$ for streams of natural numbers. The remaining corecursion structure is fixed: After producing m, we proceed corecursively with x. We cannot produce two numbers before proceeding corecursively—to do so, the blueprint would have to return $(m, (n, x)) : \mathsf{nat} \times (\mathsf{nat} \times \alpha)$.

Our first strengthening of the corecursor allows an arbitrary number of constructors before proceeding corecursively. This process takes a codatatype J and produces an initial corecursion state $\langle \mathcal{F}, \Sigma_{\mathcal{F}}, \mathsf{corec}_{\mathcal{F}} \rangle$, where $\mathcal{F}$ is a set of known friends, $\Sigma_{\mathcal{F}}$ is a BNF that incorporates the type signatures of known friends, and $\mathsf{corec}_{\mathcal{F}}$ is a corecursor. We omit the set-of-friends index whenever it is clear from the context. The initial state knows only one friend, ctor.

$$\begin{array}{l} \textsc{Base}:\quad \mathsf{J} \rightsquigarrow \langle \mathcal{F}, \Sigma_{\mathcal{F}}, \mathsf{corec}_{\mathcal{F}} \rangle \text{ where} \\ \qquad \mathcal{F} = \{\mathsf{ctor}\} \qquad \alpha\ \Sigma_{\mathcal{F}} = \alpha\ \mathsf{K_{ctor}} \qquad \mathsf{corec}_{\mathcal{F}} : (\alpha \to \alpha\ \Sigma^{+}_{\mathcal{F}}) \to \alpha \to \mathsf{J} \end{array}$$

Let us define the type $\alpha\ \Sigma^{+}_{\mathcal{F}}$ used for the corecursor. First, we let $\alpha\ \Sigma^{*}_{\mathcal{F}}$ be the free monad of Σ extended with J-constant leaves:

$\mathtt{datatype}\ \alpha\ \Sigma^{*}_{\mathcal{F}} = \mathsf{Oper}\ ((\alpha\ \Sigma^{*}_{\mathcal{F}})\ \Sigma_{\mathcal{F}}) \mid \mathsf{Var}\ \alpha \mid \mathsf{Cst}\ \mathsf{J}$

Inhabitants of $\alpha\ \Sigma^{*}_{\mathcal{F}}$ are (*formal*) *expressions* built from variable or constant leaf nodes (Var or Cst) and a syntactic representation of the constants in $\mathcal{F}$. Writing $\boxed{\mathsf{ctor}}$ for $\mathsf{Oper} : (\alpha\ \Sigma^{*}_{\mathcal{F}})\ \mathsf{K_{ctor}} \to \alpha\ \Sigma^{*}_{\mathcal{F}}$, we can build expressions such as $\boxed{\mathsf{ctor}}\ (1, \mathsf{Var}\ (x : \alpha))$ and $\boxed{\mathsf{ctor}}\ (2, \boxed{\mathsf{ctor}}\ (3, \mathsf{Cst}\ (xs : \mathsf{J})))$. The type $\alpha\ \Sigma^{+}_{\mathcal{F}}$, of *guarded expressions*, is similar to $\alpha\ \Sigma^*_{\mathcal{F}}$, except that it requires at least one $\boxed{\mathsf{ctor}}$ guard on every path to a Var. Formally, $\alpha\ \Sigma^+_{\mathcal{F}}$ is defined as $((\alpha\ \Sigma^*_{\mathcal{F}})\ \mathsf{K}_{\mathsf{ctor}})\ \Sigma^*_{\mathcal{F}}$, so that $\mathsf{K}_{\mathsf{ctor}}$ marks the guards. To simplify notation, we will pretend that $\alpha\ \Sigma^+_{\mathcal{F}} \subseteq \alpha\ \Sigma^*_{\mathcal{F}}$.

Guarded variable leaves represent corecursive calls. Constant leaves allow us to stop the corecursion with an immediate result of type J. The polymorphism of Σ^* is crucial. If we instantiate α to J, we can evaluate formal expressions with the function $\mathsf{eval} : \mathsf{J}\ \Sigma^* \to \mathsf{J}$ given by $\mathsf{eval}\ (\boxed{\mathsf{ctor}}\ x) = \mathsf{ctor}\ (\mathsf{map}_{\mathsf{K}_{\mathsf{ctor}}}\ \mathsf{eval}\ x)$, $\mathsf{eval}\ (\mathsf{Var}\ t) = t$, and $\mathsf{eval}\ (\mathsf{Cst}\ t) = t$. We also write eval for other versions of the operator (e.g., for $\mathsf{J}\ \Sigma^+$).

The corecursor's argument, the blueprint, returns guarded expressions consisting of one or more applications of $\boxed{\mathsf{ctor}}$ before proceeding corecursively. Proceeding corecursively means applying the corecursor to all variable leaves and evaluating the resulting expression. Formally:

$$\mathsf{corec}_{\mathcal{F}}\ b = \mathsf{eval} \circ \mathsf{map}_{\Sigma^+_{\mathcal{F}}}\ (\mathsf{corec}_{\mathcal{F}}\ b) \circ b$$

3.4 Adding New Friends

Corecursors can be strengthened to allow friendly functions to surround the context of the corecursive call. At the low level, we consider only uncurried functions.

A function $\mathsf{f} : \mathsf{J}\ \mathsf{K}_{\mathsf{f}} \to \mathsf{J}$ is friendly if it consumes at most one constructor before producing at least one constructor. Friendliness is captured by a mixture of two syntactic constraints and the semantic requirement of parametricity of a certain term, called the *surface*. The syntactic constraints amount to requiring that f is *expressible* using $\mathsf{corec}_{\mathcal{F}}$, irrespective of its actual definition.

Specifically, f must be equal to $\mathsf{corec}_{\mathcal{F}}\ b$ for some blueprint $b : \mathsf{J}\ \mathsf{K}_{\mathsf{f}} \to (\mathsf{J}\ \mathsf{K}_{\mathsf{f}})\ \Sigma^+$ that has the guarding constructor at the outermost position, and this object must be decomposable as $b = s \circ \mathsf{map}_{\mathsf{K}_{\mathsf{f}}} \langle \mathsf{id}, \mathsf{dtor} \rangle$ for some $s : (\alpha \times \alpha\ \mathsf{K}_{\mathsf{ctor}})\ \mathsf{K}_{\mathsf{f}} \to \alpha\ \Sigma^+$. The convolution operator $\langle f, g \rangle : \alpha \to \beta \times \gamma$ combines two functions $f : \alpha \to \beta$ and $g : \alpha \to \gamma$.

We call s the *surface* of b because it captures b's superficial layer while abstracting the application of the destructor. The surface s is more polymorphic than needed by the equation it has to satisfy. Moreover, s must be parametric in α. The decomposition, together with parametricity, ensures that friendly functions apply dtor at most once to their arguments and do not look any deeper—the "consumes at most one constructor" property.

$$\begin{array}{l} \textsc{Step}: \quad \langle \mathcal{F}, \Sigma_{\mathcal{F}}, \mathsf{corec}_{\mathcal{F}} \rangle \text{ and } \mathsf{f} : \mathsf{J}\ \mathsf{K}_{\mathsf{f}} \to \mathsf{J} \text{ friendly } \rightsquigarrow \langle \mathcal{F}', \Sigma_{\mathcal{F}'}, \mathsf{corec}_{\mathcal{F}'} \rangle \text{ where} \\ \qquad \mathcal{F}' = \mathcal{F} \cup \{\mathsf{f}\} \qquad \alpha\ \Sigma_{\mathcal{F}'} = \alpha\ \Sigma_{\mathcal{F}} + \alpha\ \mathsf{K}_{\mathsf{f}} \qquad \mathsf{corec}_{\mathcal{F}'} : (\alpha \to \alpha\ \Sigma^+_{\mathcal{F}'}) \to \alpha \to \mathsf{J} \end{array}$$

The return type of blueprints corresponding to $\mathsf{corec}_{\mathcal{F}'}$ is $\Sigma^+_{\mathcal{F}'}$, where $\Sigma_{\mathcal{F}'}$ extends $\Sigma_{\mathcal{F}}$ with K_{f}. The type $\Sigma^+_{\mathcal{F}'}$ allows all guarded expressions of the previous corecursor but may also refer to f. The syntactic representations $\boxed{\mathsf{g}} : \alpha\ \Sigma^*_{\mathcal{F}}\ \mathsf{K}_{\mathsf{g}} \to \alpha\ \Sigma^*_{\mathcal{F}}$ of old friends $\mathsf{g} \in \mathcal{F}$ must be lifted to the type $(\alpha\ \Sigma^*_{\mathcal{F}'})\ \mathsf{K}_{\mathsf{g}} \to \alpha\ \Sigma^*_{\mathcal{F}'}$, which is straightforward. In the sequel, we will reuse the notation $\boxed{\mathsf{g}}$ for the lifted syntactic representations. In addition to $\boxed{\mathsf{g}}$, new expressions are allowed to freely use the syntactic representation $\boxed{\mathsf{f}} : (\alpha\ \Sigma^*_{\mathcal{F}'})\ \mathsf{K}_{\mathsf{f}} \to \alpha\ \Sigma^*_{\mathcal{F}'}$ of the new friend f, defined as $\boxed{\mathsf{f}} = \mathsf{Oper} \circ \mathsf{Inr}$. Like for $\boxed{\mathsf{ctor}}$, we have $\mathsf{eval}\ (\boxed{\mathsf{f}}\ x) = \mathsf{f}\ (\mathsf{map}_{\mathsf{K}_{\mathsf{f}}}\ \mathsf{eval}\ x)$. As before, we have $\mathsf{corec}_{\mathcal{F}'}\ b = \mathsf{eval} \circ \mathsf{map}_{\Sigma^+_{\mathcal{F}'}}\ (\mathsf{corec}_{\mathcal{F}'}\ b) \circ b$.

The BASE operation can be seen as a special case of STEP applied to the friend ctor and a degenerate state EMPTY with $\mathcal{F} = \varnothing$.

Consider the corecursive specification of pointwise addition on streams of numbers, where $\alpha\ \mathsf{K}_{\mathsf{ctor}}$ is $\mathsf{nat} \times \alpha$ and $\mathsf{dtor}\ xs = (\mathsf{shd}\ xs,\ \mathsf{stl}\ xs)$:

$$xs \oplus ys = (\mathsf{shd}\ xs + \mathsf{shd}\ ys) \lhd (\mathsf{stl}\ xs \oplus \mathsf{stl}\ ys)$$

To make sense of this specification, we take $\alpha\ \mathsf{K}_{\oplus}$ to be $\alpha \times \alpha$ and define $\oplus$ as $\mathsf{corec}_{\mathcal{F}}\ b$, where the blueprint b is

$$\lambda p.\ (\mathsf{shd}\ (\mathsf{fst}\ p) + \mathsf{shd}\ (\mathsf{snd}\ p))\ \boxed{\lhd}\ \mathsf{Var}\ (\mathsf{stl}\ (\mathsf{fst}\ p),\ \mathsf{stl}\ (\mathsf{snd}\ p))$$

To register $\oplus$ as friendly, we must decompose b as $s \circ \mathsf{map}_{\mathsf{K}_{\oplus}} \langle \mathsf{id}, \mathsf{dtor} \rangle$. Expanding the definition of $\mathsf{map}_{\mathsf{K}_{\oplus}}$, we get

$$\begin{aligned} &\mathsf{map}_{\mathsf{K}_{\oplus}} \langle \mathsf{id}, \mathsf{dtor} \rangle \\ =\ &\lambda p.\ ((\mathsf{fst}\ p,\ \mathsf{dtor}\ (\mathsf{fst}\ p)),\ (\mathsf{snd}\ p,\ \mathsf{dtor}\ (\mathsf{snd}\ p))) \\ =\ &\lambda p.\ ((\mathsf{fst}\ p,\ (\mathsf{shd}\ (\mathsf{fst}\ p),\ \mathsf{stl}\ (\mathsf{fst}\ p))),\ (\mathsf{snd}\ p,\ (\mathsf{shd}\ (\mathsf{snd}\ p),\ \mathsf{stl}\ (\mathsf{snd}\ p)))) \end{aligned}$$

It is easy to see that the following term is a suitable surface s:

$$\lambda p'.\ (\mathsf{fst}\ (\mathsf{snd}\ (\mathsf{fst}\ p')) + \mathsf{fst}\ (\mathsf{snd}\ (\mathsf{snd}\ p')))\ \boxed{\lhd}\ \mathsf{Var}\ (\mathsf{snd}\ (\mathsf{snd}\ (\mathsf{fst}\ p')),\ \mathsf{snd}\ (\mathsf{snd}\ (\mathsf{snd}\ p')))$$

In Sect. 4, we will show how the system synthesizes blueprints and surfaces.

3.5 Merging Corecursion States

We have seen how to extend a corecursion state in a linear fashion by adding one friend after another. However, most formalizations are not linear. A module may import several other modules, giving rise to a directed acyclic graph of dependencies. We can reach a situation where the codatatype has been defined in module A; its corecursor has been extended with two different sets of friends $\mathcal{F}_B$ and $\mathcal{F}_C$ in modules B and C, each importing A; and finally module D, which imports B and C, requires a corecursor that mixes friends from $\mathcal{F}_B$ and $\mathcal{F}_C$. To support this scenario, we need an operation that merges two corecursion states.

$$\begin{aligned} &\textsc{Merge}:\quad \langle \mathcal{F}_1,\ \Sigma_{\mathcal{F}_1},\ \mathsf{corec}_{\mathcal{F}_1} \rangle \text{ and } \langle \mathcal{F}_2,\ \Sigma_{\mathcal{F}_2},\ \mathsf{corec}_{\mathcal{F}_2} \rangle \rightsquigarrow \langle \mathcal{F},\ \Sigma_{\mathcal{F}},\ \mathsf{corec}_{\mathcal{F}} \rangle \text{ where} \\ &\qquad \mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2 \qquad \alpha\ \Sigma_{\mathcal{F}} = \alpha\ \Sigma_{\mathcal{F}_1} + \alpha\ \Sigma_{\mathcal{F}_2} \qquad \mathsf{corec}_{\mathcal{F}} : (\alpha \to \alpha\ \Sigma^{+}_{\mathcal{F}}) \to \alpha \to \mathsf{J} \end{aligned}$$

The return type of blueprints for $\mathsf{corec}_{\mathcal{F}}$ is $\Sigma^{+}_{\mathcal{F}}$, where $\Sigma_{\mathcal{F}}$ is the sum of the two input signatures $\Sigma_{\mathcal{F}_1}$ and $\Sigma_{\mathcal{F}_2}$. By lifting the syntactic representations of old friends using overloading, we establish the invariant that for each $\mathsf{f} \in \mathcal{F}$ of a corecursor state, there is a syntactic representation $\boxed{\mathsf{f}} : \Sigma^{+}_{\mathcal{F}}\ \mathsf{K}_{\mathsf{f}} \to \Sigma^{+}_{\mathcal{F}}$. The function eval is then defined in the usual way and constitutes the main ingredient in the definition of $\mathsf{corec}_{\mathcal{F}}$ with the usual characteristic equation. For operations $\mathsf{f} \in \Sigma_{\mathcal{F}_1} \cap \Sigma_{\mathcal{F}_2}$, two syntactic representations are available; we arbitrarily choose the one inherited from $\Sigma_{\mathcal{F}_1}$.

Notice the difference between STEP and MERGE: While the former introduces a new friend after verifying a suitable condition, the latter lifts to a joint context two sets of operators known to be friendly, thus not requiring any verification.

3.6 Type Instantiation

We have so far ignored the potential polymorphism of J. Consider $\mathsf{J} = \alpha\ \mathsf{stream}$. The operations on corecursor states allow friends of type $(\alpha\ \mathsf{stream})\ \mathsf{K} \to \alpha\ \mathsf{stream}$ but not $(\mathsf{nat\ stream})\ \mathsf{K} \to \mathsf{nat\ stream}$. To allow friends for $\mathsf{nat\ stream}$, we must keep track of specialized corecursors. First, we need an operation for instantiating corecursor states.

$$\textsc{Instantiate}: \quad \langle F, \Sigma_{\mathcal{F}}, \mathsf{corec}_{\mathcal{F}} \rangle \rightsquigarrow \langle F[\overline{\sigma}/\overline{\alpha}], \Sigma_{\mathcal{F}}[\overline{\sigma}/\overline{\alpha}], \mathsf{corec}_{\mathcal{F}}[\overline{\sigma}/\overline{\alpha}] \rangle$$

Once we have derived a specific corecursor for $\mathsf{nat\ stream}$, we can extend it with friends of type $(\mathsf{nat\ stream})\ \mathsf{K} \to \mathsf{nat\ stream}$. Such friends cannot be added to the polymorphic corecursor, but the other direction works: Any friend of a polymorphic corecursor is also a friend of a specialized corecursor. Accordingly, we maintain a Pareto optimal subset of corecursor state instances $\{\langle \mathcal{F}_{\mathsf{S}}, \Sigma_{\mathcal{F}_{\mathsf{S}}}, \mathsf{corec}_{\mathcal{F}_{\mathsf{S}}} \rangle \mid \mathsf{S} \le \mathsf{J}\}$.

More specific corecursors are stored only if they have strictly more friends: For each pair of corecursor instances for S_1 and S_2 contained in the Pareto set, we have $\mathcal{F}_{\mathsf{S}_1} \supset \mathcal{F}_{\mathsf{S}_2}$ whenever $\mathsf{S}_1 < \mathsf{S}_2$. All the corecursors in the Pareto set are kept up to date. If we add a friend to a corecursor instance for S from the set via Step, it is also propagated to all instances S' of S by applying Instantiate to the output of Step and combining the result with the existing corecursor state for S' via Merge. When analyzing a user specification, `corec` selects the most specific applicable corecursor (the one with the most friends among those that have a general enough type).

Eagerly computing the entire Pareto set is exponentially expensive. Consider a codatatype $(\alpha, \beta, \gamma)\ \mathsf{J}$ and the friends

$$\mathsf{f}\ \text{for}\ (\mathsf{nat}, \beta, \gamma)\ \mathsf{J} \qquad \mathsf{g}\ \text{for}\ (\alpha, \beta :: \mathsf{ring}, \gamma)\ \mathsf{J} \qquad \mathsf{h}\ \text{for}\ (\alpha, \beta, \mathsf{bool})\ \mathsf{J}$$

The set would contain eight corecursors, each with a different subset of $\{\mathsf{f}, \mathsf{g}, \mathsf{h}\}$ as friends. (The type $\beta :: \mathsf{ring}$ is more specific than β due to the type class.) To avoid such an explosion, we settle for a lazy derivation strategy. In the above example, the corecursor for $(\mathsf{nat}, \beta :: \mathsf{ring}, \mathsf{bool})\ \mathsf{J}$, with $\mathsf{f}, \mathsf{g}, \mathsf{h}$ as friends, is derived only if a definition needs it.

3.7 Reasoning Principles

The primary activity of a working formalizer is to develop proofs. To conveniently reason about nonprimitively corecursive functions, `corec` provides two reasoning principles: coinduction up to congruence and a uniqueness theorem.

Coinduction up to Congruence Codatatypes are equipped with a coinduction principle. Coinduction reduces the task of proving equality between two inhabitants l and r of a codatatype to the task of exhibiting a relation R which relates l and r and is closed under application of destructors. A relation closed under destructors (or observations) is called a *bisimulation*. The `codatatype` command derives a plain coinduction rule. The rule for stream follows:

$$\frac{R\ l\ r \qquad \forall xs\ xs'.\ R\ xs\ xs' \longrightarrow \mathsf{shd}\ xs = \mathsf{shd}\ xs' \wedge R\ (\mathsf{stl}\ xs)\ (\mathsf{stl}\ xs')}{l = r}$$

To reason about functions that are corecursive up to a set of friends, a principle of coinduction up to congruence of friends is crucial. For a corecursor with friends $\mathcal{F}$, our tool derives a rule that is identical to the standard rule except with $R^{\mathcal{F}}\,(\mathsf{stl}\;xs)\;(\mathsf{stl}\;xs')$ instead of $R\;(\mathsf{stl}\;xs)\;(\mathsf{stl}\;xs')$, where $R^{\mathcal{F}}$ denotes the congruence closure of the relation R with respect to the friendly operations $\mathcal{F}$.

After registering a binary $\oplus$ on nat stream as friendly, the introduction rules for the inductively defined congruence closure include

$$\frac{x = x' \quad R^{\mathcal{F}}\,xs\;xs'}{R^{\mathcal{F}}\,(x \lhd xs)\;(x' \lhd xs')} \qquad \frac{R^{\mathcal{F}}\,xs\;xs' \quad R^{\mathcal{F}}\,ys\;ys'}{R^{\mathcal{F}}\,(xs \oplus ys)\;(xs' \oplus ys')}$$

Since the tool maintains a set of incomparable corecursors, there is also a set of coinduction principles and a set of sets of introduction rules. A technically subtle point is to make the proof assistant automatically choose the right rules in most situations. For this purpose, `tcorec` command orders the set of coinduction principles by increasing generality, which works well with Isabelle's philosophy of applying the first rule that matches. For example, after registering $\oplus$ as a friend, proving $l = r$ on nat stream might require the coinduction principle for nat stream, which is up to $\oplus$. than with the one for α stream. Users can simply write *coinduction rule*: stream.*coinduct*, and the appropriate rule is selected.

In some circumstances, it may be necessary to reason about the union of friends associated with several incomparable corecursors. To continue with the example from Sect. 3.6, suppose we want to prove a formula about $(\mathsf{nat}, \beta :: \mathsf{ring}, \mathsf{bool})\;\mathsf{J}$ by coinduction up to f, g, h before the corresponding corecursor has been derived. Users can derive it and the associated coinduction principle by invoking a dedicated command:

```
coinduction_upto J_natringbool: (nat, β :: ring, bool) J
```

The coinduction rule is exported as *J_natringbool.coinduct_upto*.

Uniqueness Principles It is sometimes possible to achieve better automation by employing a more specialized proof method than coinduction. Uniqueness principles exploit the property that the corecursor is the unique solution to a fixpoint equation:

$$h = \mathsf{eval} \circ \mathsf{map}_{\Sigma^+}\; h \circ b \longrightarrow h = \mathsf{corec}_{\mathcal{F}}\; b$$

This rule can be seen as a less powerful version of coinduction, where the bisimulation relation has been preinstantiated. In category-theoretic terms, the existence and uniqueness of a solution means that we maintain on J a completely iterative algebra [55] (whose signature is gradually incremented with each additional friend).

For concrete functions defined with `corec`, uniqueness rules can be made even more precise by instantiating the blueprint b. For example, the pointwise addition on streams from Sect. 3.4

```
corec ⊕ : nat stream → nat stream → nat stream where
  xs ⊕ ys = (shd xs + shd ys) ◁ (stl xs ⊕ stl ys)
```

yields the following uniqueness principle:

$$(\forall xs\ ys.\ h\ xs\ ys = (\mathsf{shd}\ xs + \mathsf{shd}\ ys) \lhd h\ (\mathsf{stl}\ xs)\ (\mathsf{stl}\ ys)) \longrightarrow h = \oplus$$

Reasoning by uniqueness is not restricted to functions defined with `corec`. Suppose $t\ \overline{x}$ is an arbitrary term depending on a list of free variables $\overline{x}$. The *corec_unique* proof method, also provided by our tool, transforms proof obligations of the form

$$(\forall \overline{x}.\ h\ \overline{x} = H\ \overline{x}\ h) \longrightarrow h\ \overline{x} = t\ \overline{x}$$

into $\forall \overline{x}.\ t\ \overline{x} = H\ \overline{x}\ t$. The higher-order functional H must be such that the equation $h\ \overline{x} = H\ \overline{x}\ h$ would be a valid `corec` specification (but without nested calls to h or unguarded calls). Internally, *corec_unique* extracts the blueprint b from $H\ \overline{x}\ h$ as if it would define h with $\mathsf{corec}_{\mathcal{F}}$ and uses the uniqueness principle for $\mathsf{corec}_{\mathcal{F}}$ instantiated with b to achieve the described transformation.

4 The High Level: From Commands to Definitions

AmiCo's two main commands `corec` (Sect. 4.1) and `friend_of_corec` (Sect. 4.2) introduce corecursive functions and register friends. We describe synthesis algorithms for any codatatype as implemented in the tool. We also show how to capture the "consumes at most one constructor, produces at least one constructor" contract of friends.

4.1 Defining Corecursive Functions

The `corec` command reduces the user's corecursive equation to non(co)recursive primitives, so as to guard against inconsistencies. To this end, the command engages in a chain of definitions and proofs. Recall the general context:

– The codatatype J is defined as a fixpoint of a type constructor $\alpha\ \mathsf{K_{ctor}}$ equipped with constructor ctor and destructor dtor.
– The current set of friends $\mathcal{F}$ contains ctor and has a signature $\Sigma_{\mathcal{F}}$ (or Σ). Each friend $\mathsf{f} \in \mathcal{F}$ of type $\mathsf{J}\ \mathsf{K_f} \to \mathsf{J}$ has a companion syntactic expression $\boxed{\mathsf{f}} : (\alpha\ \Sigma^*)\ \mathsf{K_f} \to \alpha\ \Sigma^*$.
– The corecursor up to $\mathcal{F}$ is $\mathsf{corec}_{\mathcal{F}} : (\alpha \to \alpha\ \Sigma^+) \to \alpha \to \mathsf{J}$.

In general, J may be polymorphic and f may take more than one argument, but these are minor orthogonal concerns here. As before, we write $\alpha\ \Sigma^*$ for the type of formal expressions built from α-leaves and friend symbols $\boxed{\mathsf{f}}$, and $\alpha\ \Sigma^+$ for $\boxed{\mathsf{ctor}}$-guarded formal expressions. For $\alpha = \mathsf{J}$, we can evaluate the formal expressions into elements of J, by replacing each $\boxed{\mathsf{f}}$ with f and omitting the Var and Cst constructors. Finally, we write eval for the evaluation functions of various types of symbolic expressions to J.

Consider the command

`corec` $\mathsf{g} : \mathsf{A} \to \mathsf{J}$ `where` $\mathsf{g}\ x = u_{\mathsf{g},x}$

where $u_{\mathsf{g},x} : \mathsf{J}$ is a term that may refer to g and x. The first task of `corec` is to synthesize a blueprint object $b : \mathsf{A} \to \mathsf{A}\,\Sigma^+$ such that

$$\mathsf{eval}\ (\mathsf{map}_{\Sigma^+} h\ (b\ x)) = u_{h,x} \qquad (4)$$

holds for all $h : \mathsf{A} \to \mathsf{J}$. This equation states that the synthesized blueprint must produce, by evaluation, the concrete right-hand side of the user equation. The unknown function h represents corecursive calls, which will be instantiated to g once g is defined. To the occurrences of h in $u_{h,x}$ correspond occurrences of Var in b.

Equipped with a blueprint, we define $\mathsf{g} = \mathsf{corec}_{\mathcal{F}}\ b$ and derive the user equation:

$$\begin{aligned}
\mathsf{g}\ x &= \mathsf{corec}_{\mathcal{F}}\ b\ x && \{\text{by definition of g}\}\\
&= \mathsf{eval}\ (\mathsf{map}_{\Sigma^+} (\mathsf{corec}\ b)\ (b\ x)) && \{\text{by } \mathsf{corec}_{\mathcal{F}}\text{'s equation}\}\\
&= \mathsf{eval}\ (\mathsf{map}_{\Sigma^+} \mathsf{g}\ (b\ x)) && \{\text{by definition of g}\}\\
&= u_{\mathsf{g},x} && \{\text{by equation (4) with g for } h\}
\end{aligned}$$

Blueprint Synthesis The blueprint synthesis proceeds by a straightforward syntactic analysis, similar to the one used for primitive corecursion [15]. We illustrate it with an example. Consider the definition of $\oplus$ from Sect. 3.4. Ignoring currying, the function has type $(\mathsf{nat\ stream})\ \mathsf{K}_\oplus \to \mathsf{nat\ stream}$, with $\alpha\ \mathsf{K}_\oplus = \alpha \times \alpha$. The term b is synthesized by processing the right-hand side of the corecursive equation for $\oplus$. After removing the syntactic sugar, we obtain the following term, highlighting the corecursive call:

$$\lambda p.\ (\mathsf{shd}\ (\mathsf{fst}\ p) + \mathsf{shd}\ (\mathsf{snd}\ p)) \lhd (\mathsf{stl}\ (\mathsf{fst}\ p)\ \oplus\ \mathsf{stl}\ (\mathsf{snd}\ p))$$

The blueprint is derived from this term by replacing the constructor guard $\lhd = \mathsf{ctor}_{\mathsf{stream}}$ and the friends with their syntactic counterparts and the corecursive call with a variable leaf:

$$b = \lambda p.\ (\mathsf{shd}\ (\mathsf{fst}\ p) + \mathsf{shd}\ (\mathsf{snd}\ p))\ \boxed{\lhd}\ \mathsf{Var}\ (\mathsf{stl}\ (\mathsf{fst}\ p), \mathsf{stl}\ (\mathsf{snd}\ p))$$

Synthesis will fail if after the indicated replacements the result does not have the desired type (here, $\mathsf{nat} \to \mathsf{nat}\ \Sigma^+$). If we omit '$(\mathsf{shd}\ (\mathsf{fst}\ p) + \mathsf{shd}\ (\mathsf{snd}\ p)) \lhd$' in the definition, the type of b becomes $\mathsf{nat} \to \mathsf{nat}\ \Sigma^*$, reflecting the lack of a guard. Another cause of failure is the presence of unfriendly operators in the call context. Once b has been produced, `corec` proves that $\oplus$ satisfies the user equation we started with.

Mixed Recursion–Corecursion If a self-call is not guarded, `corec` still gives it a chance, since it could be a terminating *recursive* call. As an example, the following definition computes all the odd numbers greater than 1 arising in the Collatz sequence:

`corec` $\mathsf{collatz} : \mathsf{nat} \to \mathsf{nat\ llist}$ `where`
  $\mathsf{collatz}\ n = \text{if } n \le 1 \text{ then } [\,] \text{ else if even } n \text{ then } \mathsf{collatz}\ \frac{n}{2} \text{ else } n \lhd \mathsf{collatz}\ (3 \cdot n + 1)$

The highlighted call is not guarded. Yet, it will eventually lead to a guarded call, since repeatedly halving a positive even number must at some point yield an odd number. The unguarded call yields a recursive specification of the blueprint b, which is resolved automatically by the termination prover.

By writing `corecursive` instead of `corec`, the user takes responsibility for proving termination. A manual proof was necessary for lfilter in Sect. 2.7, whose blueprint satisfies the recursion

$$b\ (P, xs) = \text{if } \forall x \in \mathsf{set}\ xs.\ \neg\ P\ x \text{ then } \boxed{[]}$$
$$\text{else if } P\ (\mathsf{lhd}\ xs) \text{ then } \mathsf{lhd}\ xs\ \boxed{\triangleleft}\ \mathsf{Var}\ (P, \mathsf{ltl}\ xs) \text{ else } b\ (P, \mathsf{ltl}\ xs)$$

Termination is shown by providing a suitable well-founded relation, which exists because $\mathsf{ltl}\ xs$ is closer than xs to the next element that satisfies the predicate P.

Like the corecursive calls, the recursive calls may be surrounded only by friendly operations (or by parametric operators such as ‘case’, ‘if’, and ‘let’). Thus, the following specification is rejected—and rightly so, since the unfriendly stl cancels the corecursive guard that is reached when recursion terminates.

```
corec collapz : nat → nat llist where collapz $n$ =
  if $n = 0$ then [] else if even $n$ then stl (collapz $\frac{n}{2}$) else $n \triangleleft$ collapz $(3 \cdot n + 1)$
```

We use $(\alpha,\ \alpha)\,\Sigma^\star$, where the bifunctor $(\alpha,\ \beta)\,\Sigma^\star$ abbreviates $(\alpha\ \Sigma^+ + \beta\ \Sigma^*)\ \Sigma^*$, to represent the type of formal expressions where certain subexpressions are syntactically guarded ($\alpha\ \Sigma^+$) and others are not ($\alpha\ \Sigma^*$). The first goal is to find a *pre-blueprint* $b_0 : \mathsf{A} \to (\mathsf{A},\ \mathsf{A})\ \Sigma^\star$ such that equation (4) holds, but replacing Σ^+ with the more flexible type constructor $\lambda\alpha.\ (\alpha,\ \alpha)\ \Sigma^\star$.The hope is that b_0 can give rise, by iteration, to a suitable $b : \mathsf{A} \to \mathsf{A}\ \Sigma^+$; in other words, b could emerge by the iterative application of b_0 to the unguarded components of its output until a fully guarded expression is reached. Formally, this means solving the fixpoint equation

$$b = \mathsf{flat} \circ \mathsf{map}_{\Sigma^\star}\ \mathsf{id}\ b \circ b_0 \tag{5}$$

where $\mathsf{flat} : (\alpha,\ \alpha\ \Sigma^+)\ \Sigma^\star \to \alpha\ \Sigma^+$ is the polymorphic function that merges the different layers of nesting while keeping the guard. The tool attempts to solve this fixpoint equation automatically, using Isabelle’s termination prover [19]. Alternatively, users can provide their own proof of termination. Unlike for plain corecursion, b does not satisfy the universally quantified equation (4). Instead, corec defines $\mathsf{g} = \mathsf{corec}_{\mathcal{F}}\ b$ and proves equation (4) with g for h.

The fixpoint property of b is derived from that of b_0. The construction is structured as a chase of the diagram shown in Figure 1. We must show the commutativity of the larger diagram (involving the blueprint b, which returns guarded expressions), starting from the commutativity of the leftmost quadrilateral ① (involving the pre-blueprint b_0, which returns possibly unguarded expressions). The larger diagram is filled starting from ① and constructing all the inner quadrilaterals. These inner diagrams are all commutative: ① by the choice of b_0; ② since b is a fixpoint of (5); ③ by the functoriality of map functions and the characteristic property of the corecursor; ④ by parametricity

$$\begin{tikzcd}
\mathsf{A} \arrow[rrr, "b"] \arrow[dr, "b_0"'] \arrow[ddd, "\lambda x.\, u_{\mathsf{g},x}"'] & & & \mathsf{A}\,\Sigma^{+} \arrow[ddd, "\mathsf{map}_{\Sigma^{+}}\ \mathsf{g}"] \\
& (\mathsf{A},\ \mathsf{A})\,\Sigma^{\star} \arrow[r, "\mathsf{map}_{\Sigma^{\star}}\ \mathsf{id}\ b"] \arrow[d, "\mathsf{map}_{\Sigma^{\star}}\ \mathsf{g}\ \mathsf{g}"'] \arrow[ul, phantom, "\textcircled{1}"] & (\mathsf{A},\ \mathsf{A}\,\Sigma^{+})\,\Sigma^{\star} \arrow[ur, "\mathsf{flat}"'] \arrow[d, "\mathsf{map}_{\Sigma^{\star}}\ \mathsf{g}\ (\mathsf{map}_{\Sigma^{+}}\ \mathsf{g})"'] \arrow[ul, phantom, "\textcircled{2}"] \arrow[dl, phantom, "\textcircled{3}"] \arrow[ur, phantom, "\textcircled{4}"] & \\
& (\mathsf{J},\ \mathsf{J})\,\Sigma^{\star} \arrow[dl, "\mathsf{eval}"'] & (\mathsf{J},\ \mathsf{J}\,\Sigma^{+})\,\Sigma^{\star} \arrow[l, "\mathsf{map}_{\Sigma^{\star}}\ \mathsf{id}\ \mathsf{eval}"'] \arrow[dr, "\mathsf{flat}"] \arrow[dl, phantom, "\textcircled{5}"] & \\
\mathsf{J} & & & \mathsf{J}\,\Sigma^{+} \arrow[lll, "\mathsf{eval}"]
\end{tikzcd}$$

Figure 1: Mixed recursion–corecursion

of flat; and ⑤ by invariance of evaluation under flattening. Equationally:

$$\begin{aligned}
& \mathsf{g}\ x \\
={}& \mathsf{corec}_{\mathcal{F}}\ b\ x && \{\text{by definition of } \mathsf{g}\} \\
={}& \mathsf{eval}\ (\mathsf{map}_{\Sigma^{+}}\ (\mathsf{corec}\ b)\ (b\ x)) && \{\text{by } \mathsf{corec}_{\mathcal{F}}\text{'s equation}\} \\
={}& \mathsf{eval}\ (\mathsf{map}_{\Sigma^{+}}\ \mathsf{g}\ (b\ x)) && \{\text{by definition of } \mathsf{g}\} \\
={}& \mathsf{eval}\ (\mathsf{map}_{\Sigma^{+}}\ \mathsf{g}\ (\mathsf{flat}\ (\mathsf{map}_{\Sigma^{\star}}\ \mathsf{id}\ b\ (b_0\ x)))) && \{\text{by equation (5)}\} \\
={}& \mathsf{eval}\ (\mathsf{flat}\ (\mathsf{map}_{\Sigma^{\star}}\ \mathsf{g}\ (\mathsf{map}_{\Sigma^{+}}\ \mathsf{g})\ (\mathsf{map}_{\Sigma^{\star}}\ \mathsf{id}\ b\ (b_0\ x)))) && \{\text{by parametricity of } \mathsf{flat}\} \\
={}& \mathsf{eval}\ (\mathsf{map}_{\Sigma^{\star}}\ \mathsf{id}\ \mathsf{eval}\ (\mathsf{map}_{\Sigma^{\star}}\ \mathsf{g}\ (\mathsf{map}_{\Sigma^{+}}\ \mathsf{g})\ (\mathsf{map}_{\Sigma^{\star}}\ \mathsf{id}\ b\ (b_0\ x)))) \\
& && \{\text{by invariance of evaluation under flattening}\} \\
={}& \mathsf{eval}\ (\mathsf{map}_{\Sigma^{\star}}\ \mathsf{g}\ (\mathsf{eval} \circ \mathsf{map}_{\Sigma^{+}}\ \mathsf{g} \circ b)\ (b_0\ x)) && \{\text{by functoriality of } \mathsf{map}_{\Sigma^{\star}}\} \\
={}& \mathsf{eval}\ (\mathsf{map}_{\Sigma^{\star}}\ \mathsf{g}\ \mathsf{g}\ (b_0\ x)) && \{\text{by definition of } \mathsf{g} \text{ and } \mathsf{corec}_{\mathcal{F}}\text{'s equation}\} \\
={}& u_{\mathsf{g},x} && \{\text{by the choice of } b_0\}
\end{aligned}$$

4.2 Registering New Friendly Operations

The command

`corec (friend)` $\mathsf{g} : \mathsf{J}\ \mathsf{K} \to \mathsf{J}$ `where` $\mathsf{g}\ x = u_{\mathsf{g},x}$

defines g and registers it as a friend. The domain is viewed abstractly as a type constructor K applied to the codatatype J.

The command first synthesizes the blueprint $b : \mathsf{J}\ \mathsf{K} \to \mathsf{J}\ \Sigma^{+}$, similarly to the case of plain corecursive definitions. However, this time the type Σ is not $\Sigma_{\mathcal{F}}$, but $\Sigma_{\mathcal{F}} + \mathsf{K}$. Thus, Σ^{+} mixes freely the type K with the components K_{f} of $\Sigma_{\mathcal{F}}$, which caters for *self-friendship* (as in the bfs_2 example from Sect. 2.4): g can be defined making use of itself as a friend (in addition to the already registered friends).

The next step is to synthesize a surface s from the blueprint b. Recall from Sect. 3.4 that a corecursively defined operator is friendly if its blueprint b can be decomposed as $s \circ \mathsf{map}_{\mathsf{K}}\langle \mathsf{id}, \mathsf{dtor}\rangle$, where $s : (\alpha \times \alpha\ \mathsf{K}_{\mathsf{ctor}})\ \mathsf{K} \to \alpha\ \Sigma^{+}$ is parametric in α.

Once the surface s has been synthesized, proved parametric, and proved to be in the desired relationship with b, the tool invokes the STEP operation (Sect. 3.4), enriching the corecursion state with the function defined by b as a new friend, called g.

Alternatively, users can register arbitrary functions as friends:

$\texttt{friend\_of\_corec}\ \mathsf{g} : \mathsf{J}\ \mathsf{K} \to \mathsf{J}\ \texttt{where}\ \mathsf{g}\ x = u_{\mathsf{g},x}$

The user must then prove the equation $\mathsf{g}\ x = u_{\mathsf{g},x}$. The command extracts a blueprint from it and proceeds with the surface synthesis in the same way as `corec` (`friend`).

Surface Synthesis Algorithm The synthesis of the surface from the blueprint proceeds by the context-dependent replacement of some constants with terms. AmiCo performs the replacements in a logical-relation fashion, guided by type inference.

We start with $b : \mathsf{J}\ \mathsf{K} \to \mathsf{J}\ \Sigma^+$ and need to synthesize $s : (\alpha \times \alpha\ \mathsf{K_{ctor}})\ \mathsf{K} \to \alpha\ \Sigma^+$ such that s is parametric in α and $b = s \circ \mathsf{map_K}\ \langle \mathsf{id}, \mathsf{dtor} \rangle$. We traverse b recursively and collect context information about the appropriate replacements.

Consider the definition of a function that interleaves a nonempty list of (monomorphic) streams:

$\texttt{corec}\ (\texttt{friend})\ \mathsf{inter} : \mathsf{stream\ nelist} \to \mathsf{stream}\ \texttt{where}$
$\quad \mathsf{inter}\ xss = \mathsf{shd}\ (\mathsf{hd}\ xss) \lhd \mathsf{inter}\ (\mathsf{tl}\ xss \rhd \mathsf{stl}\ (\mathsf{hd}\ xss))$

Here, β nelist is the type of nonempty lists with head and tail selectors $\mathsf{hd} : \beta\ \mathsf{nelist} \to \beta$ and $\mathsf{tl} : \beta\ \mathsf{nelist} \to \beta\ \mathsf{list}$ and $\rhd : \beta\ \mathsf{list} \to \beta \to \beta\ \mathsf{nelist}$ is defined such that $xs \rhd y$ appends y to xs. We have $\mathsf{J} = \mathsf{stream}$ (actually, $\mathsf{nat\ stream}$) and $\mathsf{K} = \mathsf{nelist}$. The blueprint is

$$b = \lambda xss.\ \mathsf{shd}\ (\mathsf{hd}\ xss)\ \boxed{\lhd}\ \mathsf{Var}\ (\mathsf{tl}\ xss \rhd \mathsf{stl}\ (\mathsf{hd}\ xss))$$

From this, the tool synthesizes the surface

$$s = \lambda xss'.\ (\mathsf{fst} \circ \mathsf{snd})\ (\mathsf{hd}\ xss') \boxed{\lhd}\ \mathsf{Var}\ ((\mathsf{map_{list}}\ \mathsf{fst} \circ \mathsf{tl})\ xss' \rhd (\mathsf{snd} \circ \mathsf{snd})\ (\mathsf{hd}\ xss'))$$

When transforming the blueprint $b : \mathsf{stream\ nelist} \to \mathsf{stream}\ \Sigma^+$ into the surface $s : (\alpha \times (\mathsf{nat} \times \alpha))\ \mathsf{nelist} \to \alpha\ \Sigma^+$, the selectors shd and stl are replaced by suitable compositions. One of the other constants, tl, is composed with a mapping of fst. The treatment of constants is determined by their position relative to the input variables (here, xss) and by whether the input is eventually consumed by a destructor-like operator on J (here, shd and stl). Bindings can also carry consumption information—from the outer context to within their scope—as in the following variant of inter:

$\texttt{corec}\ (\texttt{friend})\ \mathsf{inter'} : \mathsf{stream\ nelist} \to \mathsf{stream}\ \texttt{where}$
$\quad \mathsf{inter'}\ xss = \mathsf{case}\ \mathsf{hd}\ xss\ \mathsf{of}\ x \lhd xs \Rightarrow x \lhd \mathsf{inter'}\ (\mathsf{tl}\ xss \rhd xs)$

The case expression is syntactic sugar for a $\mathsf{case_{stream}}$ combinator. The desugared blueprint and surface constants are

$$b = \lambda xss.\ \mathsf{case_{stream}}\ (\mathsf{hd}\ xss)\ (\lambda x\, xs.\ x\ \boxed{\lhd}\ \mathsf{Var}\ (\mathsf{tl}\ xss \rhd xs))$$
$$s = \lambda xss'.\ (\mathsf{case_{prod}} \circ \mathsf{snd})\ (\mathsf{hd}\ xss')\ (\lambda x' xs'.\ x'\ \boxed{\lhd}\ \mathsf{Var}\ ((\mathsf{map_{list}}\ \mathsf{fst} \circ \mathsf{tl})\ xss' \rhd xs'))$$

The case operator for streams is processed specially, because just like shd and stl it consumes the input. The expression in the scope of the inner λ of the blueprint contains two variables—xss and xs—that have stream in their type. Due to the outer context, they must be treated differently: xss as an unconsumed input (which tells us to process the surrounding constant tl) and xs as a consumed input (which tells us to leave the surrounding constant $\triangleright$ unchanged). The selectors and case operators for J can also be applied indirectly, via mapping (e.g., $\mathsf{map}_{\mathsf{nelist}}\ \mathsf{stl}\ xss$).

Technically, surface synthesis works as follows. We generalize the relation between b and s. We fix α and abbreviate $\alpha \times \alpha\ \mathsf{K}_{\mathsf{ctor}}$ by $\hat{\alpha}$. The notion of a type σ being *compatible* with a type τ, written $\sigma \sim \tau$, and, for compatible σ, τ, the relation $\rightsquigarrow_{\sigma,\tau} : \sigma \to \tau[\mathsf{J}/\alpha] \to \mathsf{bool}$ are defined inductively:

– $\mathsf{J} \sim \alpha$ and $b \rightsquigarrow_{\mathsf{J},\alpha} b$ for all $b : \mathsf{J}$;
– $\mathsf{J} \sim \hat{\alpha}$ and $b \rightsquigarrow_{\mathsf{J},\hat{\alpha}} \langle \mathsf{id}, \mathsf{dtor}\rangle\ b$ for all $b : \mathsf{J}$;
– if $\beta \neq \alpha$, then $\beta \sim \beta$ and $b \rightsquigarrow_{\beta,\beta} b$ for all $b : \beta$;
– for an n-ary type constructor L and n-ary tuples of types $\bar{\sigma} = (\sigma_1, \ldots, \sigma_n)$ and $\bar{\tau} = (\tau_1, \ldots, \tau_n)$, if $\sigma_i \sim \tau_i$ for all $i \in \{1 \cdots n\}$, then $\bar{\sigma}\ \mathsf{L} \sim \bar{\tau}\ \mathsf{L}$ and $b \rightsquigarrow_{\bar{\sigma}\,\mathsf{L},\bar{\tau}\,\mathsf{L}} s \Leftrightarrow \mathsf{rel}_{\mathsf{L}}\ (\rightsquigarrow_{\sigma_1,\tau_1}) \ldots (\rightsquigarrow_{\sigma_n,\tau_n})\ b\ s$.

Thus, σ is compatible with τ if τ is obtained from σ by replacing occurrences of J with α or $\hat{\alpha}$. And $\rightsquigarrow_{\sigma,\tau}$ is defined by lifting, via the relator structure of the type constructors in σ and τ, the relations that trace these replacements after substituting J for α in τ: the equality relation for J versus α and the graph of $\langle \mathsf{id}, \mathsf{dtor}\rangle$ for J versus $\hat{\alpha}$.

The compatibility relation is designed to hold between the blueprint's type $\mathsf{J}\ \mathsf{K} \to \mathsf{J}\ \Sigma^+$ and the surface's type $\hat{\alpha}\ \mathsf{K} \to \alpha\ \Sigma^+$. The characteristic equation $b = s \circ \mathsf{map}_{\mathsf{K}}\ \langle \mathsf{id}, \mathsf{dtor}\rangle$ ensures that $b \rightsquigarrow_{\sigma,\tau} s$, where s denotes the $\tau[\mathsf{J}/\alpha]$ instance of the polymorphic object s. Accordingly, our synthesis problem can be generalized to the following:

Given $\sigma \sim \tau$ and $b : \sigma$, find a term $s : \tau$ such that s is parametric in α and $b \rightsquigarrow_{\sigma,\tau} s$.

The requirement on parametricity is trivially fulfilled if s does not depend on α.

When processing subterms of b under λ-abstractions, we must reason in typing contexts Γ. When processing application subterms, we lose some information about the target type τ. Factoring these constraints in, we implement the syntax-directed inference system shown in Figure 2. Judgments have the form $(\Gamma, b, \sigma) \Downarrow_{\tau_?} (\Gamma', s, \tau)$, where Γ, b, σ, Γ', and $\tau_?$ are inputs whereas s and τ are outputs. The label $\tau_?$ on the arrow represents the current partial knowledge about τ. Formally, $\tau_?$ is a type that may contain zero or more occurrences of a special placeholder ?, each representing an unknown type. We write $\tau \leq_? \tau_?$ to express that τ is consistent with the partial knowledge $\tau_?$—that is, τ is obtained from $\tau_?$ by substituting actual, potentially different types for the different occurrences of ?. We also extend the compatibility relation to allow types containing ?, by adding the inductive clause $\sigma \sim ?$, meaning that everything is compatible with ?.

We assume a partition of variables in two infinite sets and an injective function $x \mapsto x'$ between the sets. Given two contexts $\Gamma = (x_1 : \sigma_1, \ldots, x_n : \sigma_n)$ and $\Gamma' = (x'_1 : \tau_1, \ldots, x'_n : \tau_n)$, let $\Gamma \sim \Gamma'$ stand for $\forall i \in \{1, \ldots, n\}.\ \sigma_i \sim \tau_i$ and $\Gamma \rightsquigarrow \Gamma'$ stand for $\forall i \in \{1, \ldots, n\}.\ x_i \rightsquigarrow_{\sigma_i,\tau_i} x'_i$. The term $\lambda\Gamma'.\ s$ binds all the typed variables from Γ' in s.

$$\frac{x : \sigma \in \Gamma \quad x' : \tau \in \Gamma' \quad \tau \leq_? \tau_?}{(\Gamma, x, \sigma) \Downarrow_{\tau_?} (\Gamma', x', \tau)}\textsc{Var} \qquad \frac{\mathsf{lookup\_dtr}_{\mathsf{J}}\,(\mathsf{c}, \sigma) = (s, \tau) \quad \tau \leq_? \tau_?}{(\Gamma, \mathsf{c}, \sigma) \Downarrow_{\tau_?} (\Gamma', s, \tau)}\textsc{Dtr}$$

$$\frac{\mathsf{c} : \sigma(\overline{\beta}, \overline{\beta}, \overline{\gamma}) \quad \mathsf{c}\text{ parametric in }\overline{\beta}, \overline{\gamma} \quad \sigma(\hat{\alpha}, \alpha, \alpha) \leq_? \tau_?}{(\Gamma, \mathsf{c}, \sigma(\mathsf{J}, \mathsf{J}, \mathsf{J})) \Downarrow_{\tau_?} (\Gamma', \mathsf{map}_{\sigma(\hat{\alpha}, \_, \alpha)} \mathsf{fst}\ \mathsf{c}, \sigma(\hat{\alpha}, \alpha, \alpha))}\textsc{Cst}$$

$$\frac{\forall x \in \mathsf{FVars}(b).\, \forall \tau.\ x : \tau \in \Gamma \longrightarrow x' : \tau \in \Gamma' \quad \sigma \leq_? \tau_?}{(\Gamma, b, \sigma) \Downarrow_{\tau_?} (\Gamma', b, \sigma)}\textsc{Iden}$$

$$\frac{((\Gamma, x : \sigma_1), b, \sigma_2) \Downarrow_{\tau_{2?}} ((\Gamma', x' : \tau_1), s, \tau_2)}{(\Gamma, \lambda x : \sigma_1.\, b, \sigma_1 \to \sigma_2) \Downarrow_{\tau_1 \to \tau_{2?}} (\Gamma', \lambda x' : \tau_1.\, s, \tau_1 \to \tau_2)}\textsc{Abs}$$

$$\frac{\Gamma \vdash b_1 : \sigma_2 \to \sigma \quad (\Gamma, b_1, \sigma_2 \to \sigma) \Downarrow_{? \to \tau_?} (\Gamma', s_1, \tau_2 \to \tau) \quad (\Gamma, b_2, \sigma_2) \Downarrow_{\tau_2} (\Gamma', s_2, \tau_2)}{(\Gamma, b_1\ b_2, \sigma) \Downarrow_{\tau_?} (\Gamma', s_1\ s_2, \tau)}\textsc{App}_{\to}\ [\alpha \notin \sigma_2]$$

$$\frac{\Gamma \vdash b_1 : \sigma_2 \to \sigma \quad (\Gamma, b_1, \sigma_2 \to \sigma) \Downarrow_{\tau_2 \to \tau_?} (\Gamma', s_1, \tau_2 \to \tau) \quad (\Gamma, b_2, \sigma_2) \Downarrow_{?} (\Gamma', s_2, \tau_2)}{(\Gamma, b_1\ b_2, \sigma) \Downarrow_{\tau_?} (\Gamma', s_1\ s_2, \tau)}\textsc{App}_{\leftarrow}\ [\alpha \notin \sigma_2]$$

Figure 2: Inference rules for surface synthesis

Theorem 1. Assume α does not occur in Γ or σ, $\Gamma \vdash b : \sigma$, $\Gamma \sim \Gamma'$, $\sigma \sim \tau_?$ and $(\Gamma, b, \sigma) \Downarrow_{\tau_?} (\Gamma', s, \tau)$. Then, **(1)** $\Gamma' \vdash s : \tau$, and **(2)** $\lambda\Gamma'.\, s$ is parametric in α, and **(3)** $\tau \leq_? \tau_?$, and **(4)** $\sigma \sim \tau$, and **(5)** $\Gamma \rightsquigarrow \Gamma' \longrightarrow b \rightsquigarrow_{\sigma,\tau} s$.

Proof. The statements (1)–(5) are proved together by induction on the definition of $(\Gamma, b, \sigma) \Downarrow_{\tau_?} (\Gamma', s, \tau)$. We distinguish several cases, depending on which rule has been applied last:[7]

VAR: Assume $x : \sigma \in \Gamma$, $x' : \tau \in \Gamma'$, and $\tau \leq_? \tau_?$. We prove the five properties:

(1) We need that $\Gamma' \vdash x' : \tau$, which follows from $x' : \tau \in \Gamma'$.
(2) $\lambda\Gamma'.\, x'$ is parametric in α, since it is a projection function.
(3) $\tau \leq_? \tau_?$ is already an assumption.
(4) $\sigma \sim \tau$ follows from $\Gamma \sim \Gamma'$ and the definition of $\sim$, knowing that $x : \sigma \in \Gamma$ and $x' : \tau \in \Gamma'$.
(5) The implication $\Gamma \rightsquigarrow \Gamma' \longrightarrow x \rightsquigarrow_{\sigma,\tau} x'$ holds because $x : \sigma \in \Gamma$ and $x' : \tau \in \Gamma'$.

CST: Assume $\mathsf{c} : \sigma(\overline{\beta}, \overline{\beta}, \overline{\gamma})$, c is parametric in $\overline{\beta}, \overline{\gamma}$ and $\sigma(\hat{\alpha}, \alpha, \alpha) \leq_? \tau_?$. We prove the five properties:

(1) We need that $\mathsf{c} : \sigma(\hat{\alpha}, \alpha, \alpha)$ is a valid instance of c. This is ensured by the general type of c.

[7] The most interesting cases in the proof are the base cases CST and DTR, whose justifications essentially follow the explanations given for these rules in the main paper. The recursive cases, namely, ABS, APP$_{\to}$, and APP$_{\to}$, follow fairly routinely from the definition of the function space relator, $\mathsf{rel}_{\to}$. However, having the theorem stated so that the proof goes through in the recursive cases was challenging. To this end, several strengthenings of the original synthesis problem statement were required. For example, all the cases but ABS are happy with a statement of the theorem without the hypothesis $\sigma \sim \tau_?$.

(2) The aforementioned instance of c is parametric in α because c is parametric in $\overline{\beta}, \overline{\gamma}$. Moreover, since map and fst are also suitably parametric, if follows that $\mathsf{map}_{\sigma(\hat{\alpha},\_,\alpha)}\mathsf{fst}\ \mathsf{c}$ (as well as $\lambda\Gamma'.\ \mathsf{map}_{\sigma(\hat{\alpha},\_,\alpha)}\mathsf{fst}\ \mathsf{c}$) is parametric in α.
(3) We need $\sigma(\hat{\alpha}, \alpha, \alpha) \leq_? \tau_?$, which is already an assumption.
(4) The desired fact, $\sigma(\mathsf{J}, \mathsf{J}, \mathsf{J}) \sim \sigma(\hat{\alpha}, \alpha, \alpha)$, follows immediately from the definition of $\sim$.
(5) Since the contexts Γ and Γ' are irrelevant here, we must show that $\mathsf{c} \rightsquigarrow_{\sigma(\mathsf{J},\mathsf{J},\mathsf{J}),\sigma(\hat{\alpha},\alpha,\alpha)} \mathsf{map}_{\sigma(\hat{\alpha},\_,\alpha)}\mathsf{fst}\ \mathsf{c}$.
We write $\mathsf{Gr}\ f$ for the graph of a function $f : A \to B$, regarded as a relation in $A \to B \to \mathsf{bool}$, and ';' for relational composition. Thus, we have $\mathsf{Gr}\ (g \circ f) = \mathsf{Gr}\ f\ ;\ \mathsf{Gr}\ g$. From the parametricity of c, we obtain

$$\mathsf{rel}_{\sigma(\_,\_,\mathsf{J})}\ (\mathsf{Gr}\ \langle \mathsf{id}, \mathsf{dtor}\rangle)\ (\mathsf{Gr}\ \langle \mathsf{id}, \mathsf{dtor}\rangle)\ \mathsf{c}\ \mathsf{c}$$

Moreover, using the fact that the relators extend action of the map functions, we obtain

$$\mathsf{rel}_{\sigma(\hat{\mathsf{J}},\_,\mathsf{J})}\ (\mathsf{Gr}\ \mathsf{fst})\ \mathsf{c}\ (\mathsf{map}_{\sigma(\hat{\mathsf{J}},\_,\mathsf{J})}\mathsf{fst} \circ \mathsf{c})$$

In other words, Applying the map function to a term (here, fst) has the same effect as applying the relator (via the identification of functions with their graphs). Using the last two facts and the compositionality of relators, we obtain

$$\mathsf{rel}_{\sigma(\_,\_,\mathsf{J})}\ (\mathsf{Gr}\ \langle \mathsf{id}, \mathsf{dtor}\rangle)\ (\mathsf{Gr}\ \langle \mathsf{id}, \mathsf{dtor}\rangle\ ;\ \mathsf{Gr}\ \mathsf{fst})\ \mathsf{c}\ (\mathsf{map}_{\sigma(\hat{\mathsf{J}},\_,\mathsf{J})}\mathsf{fst} \circ \mathsf{c})$$

But since $\mathsf{Gr}\ \langle \mathsf{id}, \mathsf{dtor}\rangle\ ;\ \mathsf{Gr}\ \mathsf{fst}$ is the equality relation (because $\mathsf{fst} \circ \langle \mathsf{id}, \mathsf{dtor}\rangle = \mathsf{id}$), the above becomes

$$\mathsf{rel}_{\sigma(\_,\mathsf{J},\mathsf{J})}\ (\mathsf{Gr}\ \langle \mathsf{id}, \mathsf{dtor}\rangle)\ \mathsf{c}\ (\mathsf{map}_{\sigma(\hat{\mathsf{J}},\_,\mathsf{J})}\mathsf{fst} \circ \mathsf{c})$$

which means precisely $\mathsf{c} \rightsquigarrow_{\sigma(\mathsf{J},\mathsf{J},\mathsf{J}),\sigma(\hat{\alpha},\alpha,\alpha)} \mathsf{map}_{\sigma(\hat{\alpha},\_,\alpha)}\mathsf{fst} \circ \mathsf{c}$.

DTR: All the desired facts follow routinely from the definition of $\mathsf{lookup\_dtr}_{\mathsf{J}}$.

IDEN: Assume $(*)$ $\forall x \in \mathsf{FVars}(b).\ \forall\tau.\ x : \tau \in \Gamma \longrightarrow x' : \tau \in \Gamma'$ and $\sigma \leq_? \tau_?$. We prove the five properties:

(1) From $\Gamma \vdash b : \sigma$, we obtain $\Gamma\restriction_{\mathsf{FVars}(b)} \vdash b : \sigma$, where $\Gamma\restriction_{\mathsf{FVars}(b)}$ is the restriction of Γ to $\mathsf{FVars}(b)$. With $(*)$, we obtain $\Gamma'\restriction_{\mathsf{FVars}(b)} \vdash b : \sigma$, hence $\Gamma' \vdash b : \sigma$, as desired.
(2) That $\lambda\Gamma'.\ b$ is parametric in α follows from $\lambda\Gamma.\ b$ being parametric in α by a similar argument to that of point (1).
(3) The desired fact, $\sigma \leq_? \tau_?$, is already an assumption.
(4) The desired fact is $\sigma \sim \sigma$, which holds trivially since α does not appear in σ.
(5) The desired fact, namely, that $\Gamma \rightsquigarrow \Gamma' \longrightarrow b \rightsquigarrow_{\sigma,\sigma} b$ holds, follows again from $(*)$.

ABS: Assume:
(A) α fresh for Γ and $\sigma_1 \to \sigma_2$, and
(B) $\Gamma \vdash \lambda x : \sigma_1.\ b : \sigma_1 \to \sigma_2$, and
(C) $\Gamma \sim \Gamma'$, and

(D) $\sigma_1 \to \sigma_2 \sim \tau_1 \to \tau_{2?}$, and
(E) $((\Gamma, x : \sigma_1), b, \sigma_2) \Downarrow_{\tau_{2?}} ((\Gamma', x : \tau_1), s, \tau_2)$.

From (A), we obtain
($*$) α fresh for $(\Gamma, x : \sigma_1)$ and σ_2.

From (D), we obtain
(&) $\sigma_1 \sim \tau_1$,
hence, with (C), we obtain
($**$) $(\Gamma, x : \sigma_1) \sim (\Gamma', x' : \tau_1)$.

From (D), we also obtain
($***$) $\sigma_2 \sim \tau_{2?}$.

From (B) and the inversion rule for typing abstractions, we obtain
($****$) $\Gamma, x : \sigma_1 \vdash b : \sigma_2$.

Applying the induction hypothesis (IH) to ($*$), ($**$), ($***$), ($****$), and (E), we can now verify our desired facts:

(1) By (IH) we have $\Gamma', x' : \tau_1 \vdash s : \tau_2$, hence $\Gamma' \vdash \lambda x' : \tau_1.\ s : \tau_1 \to \tau_2$, as desired.
(2) By (IH) we have $\lambda(\Gamma', x' : \tau_1).\ s$ is parametric in α, which is the same as saying that $\lambda\Gamma'.\lambda x' : \tau_1.\ s$ is parametric in α, as desired.
(3) By (IH) we have $\tau_2 \leq_? \tau_{2?}$, hence $\tau_1 \to \tau_2 \leq_? \tau_1 \to \tau_{2?}$, as desired.
(4) By (IH) we have $\sigma_2 \sim \tau_2$, which, together with (&), gives $\sigma_1 \to \sigma_2 \sim \tau_1 \to \tau_2$, as desired.
(5) Assume (#) $\Gamma \rightsquigarrow \Gamma'$. We need to show $(\lambda x : \sigma_1.\ b) \rightsquigarrow_{\sigma_1 \to \sigma_2, \tau_1 \to \tau_2} (\lambda x' : \tau_1.\ s)$. By the definitions of $\rightsquigarrow$ and of the function space relator $\mathsf{rel}_{\to}$, this means assuming ($) $x \rightsquigarrow_{\sigma_1,\tau_1} x'$ and showing $b \rightsquigarrow_{\sigma_2,\tau_2} s$. From (#) and ($), we obtain $(\Gamma, x : \sigma_1) \rightsquigarrow (\Gamma', x : \tau_1)$, which by (IH) implies $b \rightsquigarrow_{\sigma_2,\tau_2} s$, as desired.

APP$_{\to}$: Assume:
(A) α fresh for Γ and σ, and
(B) $\Gamma \vdash b_1 b_2 : \sigma$, and
(C) $\Gamma \sim \Gamma'$, and
(D) $\sigma \sim \tau_?$, and
($***$1) $\Gamma \vdash b_1 : \sigma_2 \to \sigma$, and
(E1) $(\Gamma, b_1, \sigma_2 \to \sigma) \Downarrow_{? \to \tau_?} (\Gamma', s_1, \tau_2 \to \tau)$, and
(E2) $(\Gamma, b_2, \sigma_2) \Downarrow_{\tau_2} (\Gamma', s_2, \tau_2)$, and
($*$2) α fresh for σ_2.

From (A) and ($*$2), we obtain
($*$1) α fresh for Γ and $\sigma_2 \to \sigma$.

From (D) and the definition of compatibility, we obtain
($**$1) $\sigma_2 \to \sigma \sim ? \to \tau_?$.

From (B), ($***$1), and the inversion rule for typing applications, we obtain
($***$2) $\Gamma \vdash b_2 : \sigma_2$.

We will write (IH1) to indicate the application of the induction hypothesis to ($*$1), (C), ($**$1), ($***$1), and (E1). By (IH1), we have
(&) $\sigma_2 \to \sigma \sim \tau_2 \to \tau$, hence
($**$2) $\sigma_2 \sim \tau_2$.

In addition to (IH1), now we can also apply the induction hypothesis to ($*2$), (C), ($**2$), ($***2$), and (E2), which we will refer to as (IH2). We use (IH1) and (IH2) to obtain the desired facts:

(1) By (IH1) we have $\Gamma' \vdash s_1 : \tau_2 \to \tau$; by (IH2) we have $\Gamma' \vdash s_2 : \tau_2$; from these, we obtain $\Gamma' \vdash s_1 s_2 : \tau$, as desired.
(2) By (IH1) we have that $\lambda\Gamma'.\ s_1$ is parametric in α; by (IH2) we have that $\lambda\Gamma'.\ s_2$ is parametric in α; these imply that $\lambda\Gamma'.\ s_1 s_2$ is parametric in α, as desired.
(3) By (IH1) we have $\tau_2 \to \tau \leq_? ? \to \tau_?$, hence $\tau \leq_? \tau_?$, as desired.
(4) From (&), we have $\sigma \sim \tau$, as desired.
(5) Assume (#) $\Gamma \rightsquigarrow \Gamma'$. By (IH1), we have $b_1 \rightsquigarrow_{\sigma_2\to\sigma,\tau_2\to\tau} s_1$; by (IH2), we have $b_2 \rightsquigarrow_{\sigma_2,\tau_2} s_2$; from these, using the definitions of $\rightsquigarrow$ and of the function space relator $\mathsf{rel}_\to$, we obtain $b_1 b_2 \rightsquigarrow_{\sigma,\tau} s_1 s_2$, as desired.

App$_\leftarrow$: Assume:
(A) α fresh for Γ and σ, and
(B) $\Gamma \vdash b_1 b_2 : \sigma$, and
(C) $\Gamma \sim \Gamma'$, and
(D) $\sigma \sim \tau_?$, and
($***1$) $\Gamma \vdash b_1 : \sigma_2 \to \sigma$, and
(E1) $(\Gamma, b_1, \sigma_2 \to \sigma) \Downarrow_{\tau_2\to\tau_?} (\Gamma', s_1, \tau_2 \to \tau)$, and
(E2) $(\Gamma, b_2, \sigma_2) \Downarrow_? (\Gamma', s_2, \tau_2)$, and
($*2$) α fresh for σ_2.
From (A) and ($*2$), we obtain
($*1$) α fresh for Γ and $\sigma_2 \to \sigma$.
From (D) and the definition of compatibility, we have
($**1$) $\sigma_2 \to \sigma \sim \tau_2 \to \tau_?$, and
($**2$) $\sigma_2 \sim ?$.
From (B), ($***1$) and the inversion rule for typing applications, we obtain
($***2$) $\Gamma \vdash b_2 : \sigma_2$.
Applying the induction hypothesis to ($*1$), (C), ($**1$), ($***1$), and (E1) (which we will refer to as (IH1)) on the one hand, and the induction hypothesis to ($*2$), (C), ($**2$), ($***2$), and (E2) (which we will refer to as (IH2)) on the other hand, we obtain the desired facts. Facts (1), (2), and (5) follow by (IH1) and (IH2) using an argument identical to that in the App$_\to$ case. So we only prove (3) and (4):

(3) By (IH1) we have $\tau_2 \to \tau \leq_? \tau_2 \to \tau_?$, hence $\tau \leq_? \tau_?$, as desired.
(4) By (IH1) we have $\sigma_2 \to \sigma \sim \tau_2 \to \tau$, hence $\sigma \sim \tau$, as desired. □

As a corollary, this theorem gives a sound solution to the synthesis problem, taking Γ, Γ' to be empty and $\tau_?$ to be τ:

Corollary 2 If α does not occur in σ, $\sigma \sim \tau$, $b : \sigma$, and $([], b, \sigma) \Downarrow_\tau ([], s, \tau')$, then **(1)** $\tau' = \tau$, and **(2)** $s : \tau$, and **(3)** s is parametric in α, and **(4)** $b \rightsquigarrow_{\sigma,\tau} s$.

Proof. Everything but $\tau' = \tau$ follows from the theorem, taking $\Gamma = \Gamma' = []$. Moreover, the theorem also ensures $\tau' \leq_? \tau$; this, together with the fact that τ does not contain ?, immediately implies $\tau' = \tau$ by the definition of $\leq_?$. □

$$\dfrac{\dfrac{}{\begin{array}{c}(\Gamma, \mathsf{shd}, \mathsf{stream} \to \mathsf{nat})\\ \Downarrow_{? \to ?}\\ (\Gamma', \mathsf{fst} \circ \mathsf{snd}, \hat{\alpha} \to \mathsf{nat})\end{array}}\textsc{Dtr} \qquad \dfrac{\dfrac{}{\begin{array}{c}(\Gamma, \mathsf{hd}, \mathsf{stream\ nelist} \to \mathsf{stream})\\ \Downarrow_{\hat{\alpha}\ \mathsf{nelist} \to \hat{\alpha}}\\ (\Gamma', \mathsf{hd}, \hat{\alpha}\ \mathsf{nelist} \to \hat{\alpha})\end{array}}\textsc{Cst} \quad \dfrac{}{\begin{array}{c}(\Gamma, \mathit{xss}, \mathsf{stream\ nelist})\\ \Downarrow_{?}\\ (\Gamma', \mathit{xss}', \hat{\alpha}\ \mathsf{nelist})\end{array}}\textsc{Var}}{(\Gamma, \mathsf{hd}\ \mathit{xss}, \mathsf{stream}) \Downarrow_{\hat{\alpha}} (\Gamma', \mathsf{hd}\ \mathit{xss}', \hat{\alpha})}\textsc{App}_{\leftarrow}}{(\Gamma, \mathsf{shd}\ (\mathsf{hd}\ \mathit{xss}), \mathsf{nat}) \Downarrow_{?} (\Gamma', (\mathsf{fst} \circ \mathsf{snd})\ (\mathsf{hd}\ \mathit{xss}'), \mathsf{nat})}\textsc{App}_{\to}$$

Figure 3: Surface synthesis for the subterm $\mathsf{shd}\ (\mathsf{hd}\ \mathit{xss})$ of inter's blueprint

$$\dfrac{\dfrac{}{\begin{array}{c}(\Gamma, \mathsf{ltl}, \mathsf{stream\ nelist} \to \mathsf{stream\ list})\\ \Downarrow_{\hat{\alpha}\ \mathsf{nelist} \to \alpha\ \mathsf{list}}\\ (\Gamma', \mathsf{map}_{\mathsf{list}}\mathsf{fst} \circ \mathsf{ltl}, \hat{\alpha}\ \mathsf{nelist} \to \alpha\ \mathsf{list})\end{array}}\textsc{Cst} \quad \dfrac{}{\begin{array}{c}(\Gamma, \mathit{xss}, \mathsf{stream\ nelist})\\ \Downarrow_{?}\\ (\Gamma', \mathit{xss}', \hat{\alpha}\ \mathsf{nelist})\end{array}}\textsc{Var}}{(\Gamma, \mathsf{ltl}\ \mathit{xss}, \mathsf{stream\ list}) \Downarrow_{\alpha\ \mathsf{list}} (\Gamma', (\mathsf{map}_{\mathsf{list}}\mathsf{fst} \circ \mathsf{ltl})\ \mathit{xss}', \alpha\ \mathsf{list})}\textsc{App}_{\leftarrow}$$

Figure 4: Surface synthesis for the subterm $\mathsf{ltl}\ \mathit{xss}$ of inter's blueprint

The inference system consists of two kinds of rule. The base rules VAR, DTR, CST, and IDEN actually perform the constant and variable replacements; the recursive rules ABS, APP$_{\to}$, and APP$_{\leftarrow}$ collect type information (in $\tau_?$) to guide the replacements.

ABS delves into λ-abstractions by enriching the typing context with a new pair of variables. VAR collects matching variables from the context, provided the type is consistent with the partial knowledge.

CST and DTR form the heart of the synthesis. CST processes arbitrary constants, taking advantage of any parametricity they may exhibit. In the rule, σ is polymorphic in the type variables $\overline{\beta}, \overline{\beta'}, \overline{\gamma}$. The tuples $\overline{\beta}$, $\overline{\beta'}$, and $\overline{\gamma}$ are mutually disjoint and non-repetitive, and the tuples $\overline{\beta}$ and $\overline{\beta'}$ have the same length. Thinking of σ as $\sigma(\overline{\beta}, \overline{\beta'}, \overline{\gamma})$, we write $\sigma(\overline{\beta}, \overline{\beta}, \overline{\gamma})$ for σ with the variables in $\overline{\beta'}$ replaced by those in $\overline{\beta}$, component-wise; $\sigma(\hat{\alpha}, \alpha, \alpha)$ for σ with $\overline{\beta}, \overline{\beta'}, \overline{\gamma}$ replaced by $\hat{\alpha}, \alpha, \alpha$, respectively (i.e., all variables in $\overline{\beta}$ replaced by $\hat{\alpha}$ and so on); and similarly for other types. We assume that the variables $\overline{\beta'}$ occur only positively in σ and write $\mathsf{map}_{\sigma(\hat{\alpha},\_,\alpha)}$ for the map function of type $(\alpha_1 \to \alpha_2) \to \sigma(\hat{\alpha}, \alpha_1, \alpha) \to \sigma(\hat{\alpha}, \alpha_2, \alpha)$.

Thus, $\sigma = \sigma(\overline{\beta}, \overline{\beta'}, \overline{\gamma})$ is a generalization of the type $\sigma(\overline{\beta}, \overline{\beta}, \overline{\gamma})$ of c where some positive occurrences of the type variables in $\overline{\beta}$ have been replaced by copies from $\overline{\beta'}$. The rule allows us to synthesize a term of type $\sigma(\hat{\alpha}, \alpha, \alpha)$ from the $\sigma(\mathsf{J}, \mathsf{J}, \mathsf{J})$ instance of c. If $\overline{\beta}$ (hence also $\overline{\beta'}$) is empty, we can simply synthesize the $\sigma(\hat{\alpha}, \alpha, \alpha)$ instance, because c is parametric in $\overline{\gamma}$ and all the occurrences of J corresponding to $\overline{\gamma}$ must become α. In this case, $\mathsf{c} \rightsquigarrow_{\sigma(\mathsf{J},\mathsf{J},\mathsf{J}),\sigma(\hat{\alpha},\alpha,\alpha)} \mathsf{c}$ follows immediately by parametricity.

The situation is more subtle if $\overline{\beta}$ is not empty. For each type variable β_i from $\overline{\beta}$, some of its occurrences may need to be instantiated to $\hat{\alpha}$, while others (those that become β'_i in σ) must be instantiated to α. The rule copes with this lack of uniformity by applying fst, the left inverse of $\langle \mathsf{id}, \mathsf{dtor} \rangle$, at the $\overline{\beta'}$ positions, using $\mathsf{map}_{\sigma(\hat{\alpha},\_,\alpha)}\ \mathsf{fst}$. This enforces the desired relationship: $\mathsf{c} \rightsquigarrow_{\sigma(\mathsf{J},\mathsf{J},\mathsf{J}),\sigma(\hat{\alpha},\alpha,\alpha)} \mathsf{map}_{\sigma(\hat{\alpha},\_,\alpha)}\mathsf{fst}\ \mathsf{c}$. While we require that $\overline{\beta}$ and

$\overline{\beta'}$ have the same length, not all variables β'_i need occur in σ. If β'_i does not occur in σ, $\mathsf{map}_{\sigma(\hat{\alpha},\_,\alpha)}$ is not influenced by β'_i.

Concretely, when applying CST, the parameters are instantiated by matching the type of c with the partial type information $\tau_?$. If some occurrences of a type variable δ match $\hat{\alpha}$ and others match α, we have $\delta = \beta_i$ for some i, the occurrences of δ that match $\hat{\alpha}$ are left unchanged, and the occurrences that match α become β'_i. If only $\hat{\alpha}$ is matched, then $\delta = \beta_i$ is left unchanged, and the corresponding β'_i will not occur in σ. Finally, if only α is matched, $\delta = \gamma_i$ for some i. In general, there may be no solution (resulting in a failure), or several solutions due to the unknowns ? occurring in $\tau_?$.

Figures 3 and 4 show parts of the inference tree for the blueprint of inter, with $\Gamma = (xss : \mathsf{stream\ nelist})$ and $\Gamma' = (xss' : \hat{\alpha}\ \mathsf{nelist})$. The case of $\mathsf{hd} : \beta\ \mathsf{nelist} \to \beta$ from Figure 3 is simple. We start with the stream nelist $\to$ stream instance of hd and must synthesize a term of type $\hat{\alpha}\ \mathsf{nelist} \to \hat{\alpha}$. When applying CST, we take $\sigma(\overline{\beta}, \overline{\beta'}, \overline{\gamma})$ to be $\beta\ \mathsf{nelist} \to \beta$ (i.e., $\overline{\gamma}$ is empty, $\overline{\beta}$ and $\overline{\beta'}$ are singletons, and β' does not occur in σ), making the map function $\mathsf{map}_{\sigma(\hat{\alpha},\_,\alpha)}$ an identity, which leads to the synthesis of $\mathsf{map}_{\sigma(\hat{\alpha},\_,\alpha)}\ \mathsf{fst\ hd} = \mathsf{id\ hd} = \mathsf{hd}$.

In Figure 4, the constant $\mathsf{ltl} : \beta\ \mathsf{nelist} \to \beta\ \mathsf{list}$ receives a different treatment. We start with the stream nelist $\to$ stream list instance of ltl and must synthesize a term of type $\hat{\alpha}\ \mathsf{nelist} \to \alpha\ \mathsf{list}$. To acknowledge the mismatch between $\hat{\alpha}$ and α, both of which correspond to stream in the type of ltl, $\sigma(\overline{\beta}, \overline{\beta'}, \overline{\gamma})$ is taken to be $\beta\ \mathsf{nelist} \to \beta'\ \mathsf{list}$; thus, $\overline{\gamma}$ is empty and $\overline{\beta}$ and $\overline{\beta'}$ are singletons as before, but this time β' occurs in σ. Since β' occurs positively, CST is applicable, synthesizing $\mathsf{map}_{\sigma(\hat{\alpha},\_,\alpha)}\ \mathsf{fst\ hd} = \mathsf{map}_{\mathsf{list}}\ \mathsf{fst} \circ \mathsf{ltl}$.

The DTR rule processes destructor-like operators for J specially: the destructor dtor itself, the selectors, the discriminators, and the case operator. For J = stream, we have two selectors, shd and stl, and no discriminator; for J = llist, we have lhd and ltl and an "is nil" discriminator. Isabelle's `codatatype` command generates these operators [15]. The partial function $\mathsf{lookup\_dtr}_\mathsf{J}$, referenced in DTR, maps destructor operators to surfaces (both paired with their type). For $\mathsf{dtor} : \mathsf{J} \to \mathsf{J}\ \mathsf{K}_{\mathsf{ctor}}$, $\mathsf{lookup\_dtr}_\mathsf{J}$ returns $\mathsf{snd} : \hat{\alpha} \to \alpha\ \mathsf{K}_{\mathsf{ctor}}$, acknowledging the direct consumption of a destructor. Indeed, we have $\mathsf{J} \to \mathsf{J}\ \mathsf{K}_{\mathsf{ctor}} \sim \hat{\alpha} \to \alpha\ \mathsf{K}_{\mathsf{ctor}}$ and $\mathsf{dtor} \rightsquigarrow_{\mathsf{J} \to \mathsf{J}\ \mathsf{K}_{\mathsf{ctor}}, \hat{\alpha} \to \alpha\ \mathsf{K}_{\mathsf{ctor}}} \mathsf{snd}$; hence, $\mathsf{dtor} = \mathsf{snd} \circ \langle \mathsf{id}, \mathsf{dtor} \rangle$. For the other destructor operators, $\mathsf{lookup\_dtr}_\mathsf{J}$ applies the observation that they can be seen as derived by composing dtor with a suitable parametric function. For the stream case, $\mathsf{shd} = \mathsf{fst} \circ \mathsf{dtor}$, $\mathsf{stl} = \mathsf{snd} \circ \mathsf{dtor}$, and $\mathsf{case}_{\mathsf{stream}} = \mathsf{case}_{\mathsf{prod}} \circ \mathsf{dtor}$ are transformed into $\mathsf{fst} \circ \mathsf{snd}$, $\mathsf{snd} \circ \mathsf{snd}$, and $\mathsf{case}_{\mathsf{prod}} \circ \mathsf{snd}$, respectively. Finally, for constants other than the destructor operators, $\mathsf{lookup\_dtr}_\mathsf{J}$ is not defined.

The IDEN rule allows us to stop without traversing the term: If b has no free variables whose types have changed from Γ to Γ' (in particular, if it has no free variables at all) and its type is consistent with the partial information, we synthesize b itself.

For handling application terms, we start with $b_1\ b_2$ such that $b_1 : \sigma_2 \to \sigma$ and $b_2 : \sigma_2$ and with partial knowledge $\tau_?$ of the type of the synthesized object. Two rules may apply. In $\mathrm{APP}_\to$, the information flows from the left to the right premise: First synthesize s_1 and its type $\tau_2 \to \tau_?$ starting from partial knowledge $? \to \tau_?$); then synthesize s_2, having full knowledge of its type, τ_2. By contrast, $\mathrm{APP}_\leftarrow$ operates from right to left: First synthesize s_2 and its type τ_2, starting without any knowledge of its type; then synthesize s_1 and its type starting from partial knowledge $\tau_2 \to \tau_?$.

To avoid backtracking, we can employ a strategy to choose between $\text{App}_{\rightarrow}$ and $\text{App}_{\leftarrow}$ when analyzing an application $b_1\ b_2$: If the argument b_2 is an input variable, $\text{App}_{\leftarrow}$ is applied; otherwise, $\text{App}_{\rightarrow}$ is chosen. This is complete for a large practically motivated syntactic fragment. When processing $\mathsf{shd}\ (\mathsf{hd}\ xss)$ in Figure 3, this strategy first gives priority to shd, and then to xss in $\mathsf{hd}\ xss$.

So far, we have seen how synthesis succeeds, producing a surface that is guaranteed to be correct (Theorem 1). When synthesis fails, it catches unfriendly attempts to consume more than one destructor. Suppose we want to process $\mathsf{shd}\ (\mathsf{stl}\ xs)$ with typing contexts $\Gamma = (xs : \mathsf{stream})$ and $\Gamma' = (xs' : \hat{\alpha})$. From processing shd, we obtain that $\mathsf{stl}\ xs$ must be translated to a term of type $\hat{\alpha}$, which forces $\mathsf{stl} : \mathsf{stream} \rightarrow \mathsf{stream}$ to yield something of a type $? \rightarrow \hat{\alpha}$. The rule Iden fails because $\mathsf{stream} \rightarrow \mathsf{stream} \not\lesssim_{?}\ ? \rightarrow \hat{\alpha}$. Dtr does not apply either, since the resulting type would be $\hat{\alpha} \rightarrow \alpha$, which is again inconsistent with $? \rightarrow \hat{\alpha}$. The presence of α on the right of $\hat{\alpha} \rightarrow \alpha$ signals a consumed input, contradicting the constraint $? \rightarrow \hat{\alpha}$.

5 Implementation in Isabelle/HOL

The implementation of AmiCo followed the same general strategy as that of most other definitional mechanisms for Isabelle:

1. We started from an abstract formalized example consisting of a manual construction of the Base and Step corecursors and the corresponding reasoning principles.
2. We streamlined the formal developments, eliminating about 1000 lines of Isabelle definitions and proofs—to simplify the implementation and improve performance.
3. We formalized the new Merge operation in the same style as Base and Step.
4. We developed Standard ML functions to perform the corecursor state operations for arbitrary codatatypes and friendly functions.
5. We implemented, also in Standard ML, the commands that process user specifications and interact with the corecursor state.

HOL's type system cannot express quantification over arbitrary BNFs, thus the need for ML code to repeat the corecursor derivations for each new codatatype or friend. With the foundational approach, not only the corecursors and their characteristic theorems are produced but also all the intermediate objects and lemmas, to reach the highest level of trustworthiness. Assuming the proof assistant's inference kernel is correct, bugs in our tool can lead at most to run-time failures, never to logical inconsistencies.

The code for step 4 essentially constructs the low-level types, terms, and lemma statements presented in Sect. 3 and proves the lemmas using dedicated *tactics*—ML programs that generalize the proofs from the formalization. The lemmas have a fixed shape; in principle, the tactics always succeed. The code for step 5 analyses the user's specification and synthesizes blueprints and surfaces, as described in Sect. 4. It reuses `primcorec`'s parsing combinators [15] for recognizing map functions and other syntactic conveniences, such as the use of λs as an alternative to $\circ$ for corecursing under $\rightarrow$, as seen in Sect. 2.1. Proof obligations pertaining to parametricity and termination may require custom reasoning beyond what the standard tactics offer [19, 39].

Because a constant or theorem cannot be defined twice, the implementation of STEP and MERGE associates version numbers with corecursor states. These numbers appear in the names of the generated constants and lemmas, as an optional component, next to the name of the codatatype J (e.g., J.*v3.coinduct_upto*). When the user refers to a name without a version number (e.g., J.*coinduct_upto*), it gets resolved to the highest version. In practice, users hardly ever need to refer to numbers, because they can also use the aliases generated by `corec` and `friend_of_corec` (e.g., f.*coinduct*) or by `coinduction_upto` (e.g., *J_natringbool.coinduct_upto*).

The archive accompanying this paper [14] contains instructions that explain where to find the code and the users' manual and how to run the code.

6 Related Work and Discussion

This work combines the safety of foundational approaches to function definitions with an expressive flavor of corecursion and mixed recursion–corecursion. It continues a program of integrating category theory insight into proof assistant technology [15–17, 73]. There is a lot of related work on corecursion and productivity, both theoretical and applied to proof assistants and functional programming languages.

Theory of (Co)recursion AmiCo incorporates category theory from many sources, notably Milius et al. [56] for corecursion up-to and Rot et al. [65] for coinduction up-to. Our earlier papers [16, 73] discuss further theoretical sources. AmiCo implements the first general, provably sound, and fully automatic method for mixing recursive and corecursive calls in function definitions. The idea of mixing recursion and corecursion appears in Bertot [11] for the stream filter, and a generalization is sketched in Bertot and Komendantskaya [13] for corecursion up to constructors. Leino's Dafny tool [49] was the first to offer such a mixture for general codatatypes, which turned out to be unsound and was subsequently restricted to the sound but limited fragment of tail recursion.

Corecursion in Other Proof Assistants Coq supports productivity by a syntactic guardedness check, based on the pioneering work of Giménez [28]. MiniAgda [2] and Agda implement a more flexible approach to productivity due to Abel et al. [3, 5], based on sized types and copatterns. Coq's guardedness check allows, in our terminology, only the constructors as friends [20]. By contrast, Agda's productivity checker is more expressive than AmiCo's, because sized types can capture more precise contracts than the "consumes at most one constructor, produces at least one constructor" criterion. For example, a Fibonacci stream definition such as $\mathsf{fib} = 0 \lhd 1 \lhd (\mathsf{fib} + \mathsf{stl}\ \mathsf{fib})$ can be made to work in Agda, but is rejected by AmiCo because stl is not a friend—so we need to rephrase this into a friendly definition: $\mathsf{fib} = 0 \lhd (1 \lhd \mathsf{fib} + \mathsf{fib})$. As mentioned in Sect. 2.4, this flexibility comes at a price: The user must encode the productivity argument in the function's type, leading to additional proof obligations.

In the world of algebraic specifications, CIRC [53] is a theorem prover designed for automating coinduction via sound circular reasoning. It bears similarity with both Coq's Paco and our AmiCo. Its freezing operators are an antidote to what we would call the absence of friendship: Equality is no longer a congruence, hence equational reasoning is frozen at unfriendly locations.

Foundational Function Definitions AmiCo's commands and proof methods fill a gap in Isabelle/HOL's coinductive offering. They complement `codatatype`, `primcorec`, and *coinduction* [15], allowing users to define nonprimitive corecursive and mixed recursive–corecursive functions. Being foundational, our work offers a strong protection against inconsistency by reducing circular fixpoint definitions issued by the user to low-level acyclic definitions in the core logic. This approach has a long tradition.

Most systems belonging to the HOL family include a counterpart to the `primrec` command of Isabelle, which synthesizes the argument to a primitive recursor. Isabelle/HOL is the only HOL system that also supports codatatypes and `primcorec` [15]. Isabelle/ZF, for Zermelo–Fraenkel set theory, provides `(co)datatype` and `primrec` [61] commands, but no high-level mechanisms for defining corecursive functions.

For nonprimitively recursive functions over datatypes, Slind's TFL package for HOL4 and Isabelle/HOL [68] and Krauss's `function` command for Isabelle/HOL [45] are the state of the art. Krauss developed the `partial_function` command for defining monadic functions [46]. Definitional mechanisms based on the Knaster–Tarski fixpoint theorems were also developed for (co)inductive predicates [34, 61]. HOLCF, a library for domain theory, offers a `fixrec` command for defining continuous functions [38].

Our handling of friends can be seen as a round trip between a shallow and a deep embedding that resembles normalization by evaluation [9] (but starting from the shallow side). Initially, the user specification contains shallow (semantic) friends. For identifying the involved corecursion as sound, the tool reifies the friends into deep (syntactic) friends, which make up the blueprint. Then the deep friends are "reflected" back into their shallow versions by the evaluation function $\mathsf{eval} : \mathsf{J}\,\Sigma^{*} \to \mathsf{J}$. A similar technique is used by Myreen in HOL4 for verification and synthesis of functional programs [59].

In Agda, Coq, and Matita, the definitional mechanisms for (co)recursion are built into the system. In contrast, Lean axiomatizes only the recursor and derives arbitrary specifications in the style of `primrec` to reduce the trusted code base [58]. The distinguishing features of AmiCo are its dynamicity and high level of automation. The derived corecursors and coinduction principles are updated with new ones each time a friend is registered. This permits reuse both internally (resulting in lighter constructions) and at the user level (resulting in fewer proof obligations).

Code Extraction Isabelle's code generator [32] extracts Haskell code from an executable fragment of HOL, mapping HOL (co)datatypes to lazy Haskell datatypes and HOL functions to Haskell functions. Seven out of our eight case studies fall into this fragment; the extracted code is part of the archive [14]. Only the filter function on lazy lists is clearly not computable (Sect. 2.7). In particular, extraction works for Lochbihler's probabilistic calculus (Sect. 2.8) which involves the type spmf of discrete subprobability distributions. Verified data refinement in the code generator makes it possible to implement such BNFs in terms of datatypes, e.g., spmf as associative lists similar to Erwig's and Kollmansberger's PFP library [26]. Thus, we can extract code for GPVs and their operations like inlining. Lochbihler and Züst [52] used an earlier version of the calculus to implement a core of the Transport Layer Security (TLS) protocol in HOL.

Certified Lazy Programming Our tool and the examples are a first step towards a framework for friendship-based certified programming: Programs are written in the executable fragment, verified in Isabelle, and extracted to Haskell. AmiCo ensures that corecursive definitions are productive and facilitates coinductive proofs by providing strong coinduction rules. Productivity and termination of the extracted code are guaranteed if the whole program is specified in HOL exclusively with datatypes, codatatypes, recursive functions with the `function` command, and corecursive functions with `corec`, and no custom congruence rules for higher-order operators have been used.[8]If the restrictions are met, the program clearly lies within the executable fragment and the code extracted from the definitions yields the higher-order rewrite system which the termination prover and AmiCo have checked. In particular, these restrictions exclude the noncomputable filter function on lazy lists (Sect. 2.7), with the test $\forall n \in \mathsf{set}\, xs.\ \neg\, P\, n$.

A challenge will be to extend these guarantees to Isabelle's modular architecture. Having been designed with only partial correctness in mind, the code extractor can be customized to execute arbitrary (proved) equations—which can easily break productivity and termination. A similar issue occurs with `friend_of_corec`, which cares only about semantic properties of the friend to be. For example, we can specify the identity function id on streams by $\mathsf{id}\ (x \lhd y \lhd xs) = x \lhd y \lhd xs$ and register it as a friend with the derived equation $\mathsf{id}\ x = \mathsf{shd}\ x \lhd \mathsf{stl}\ x$. Consequently, AmiCo accepts the definition $\mathsf{natsFrom}\ n = n \lhd \mathsf{id}\ (\mathsf{natsFrom}\ (n+1))$, but the extracted Haskell code diverges. To avoid these problems, we would have to (re)check productivity and termination on the equations used for extraction. In this scenario, AmiCo can be used to distinguish recursive from corecursive calls in a set of (co)recursive equations, and synthesize sufficient conditions for the function being productive and the recursion terminating, and automatically prove them (using Isabelle's parametricity [39] and termination provers [19]).

AmiCo beyond Higher-Order Logic The techniques implemented in our tool are applicable beyond Isabelle/HOL. In principle, nothing stands in the way of AgdamiCo, AmiCoq, or MatitamiCo. Danielsson [22] and Thibodeau et al. [70] showed that similar approaches work in type theory; what is missing is a tool design and implementation. AmiCo relies on parametricity, which is now understood for dependent types [10].

In Agda, parametricity could be encoded with sized types, and AgdamiCo could be a foundational tool that automatically adds suitable sized types for justifying the definition and erases them from the end product. Coq includes a parametricity-tracking tool [43] that could form the basis of AmiCoq. The Paco library by Hur et al. [40] facilitates coinductive proofs based on parameterized coinduction [57, 76]. Recent work by Pous [63] provides further advances, including a framework to combine proofs by

[8] The `function` command uses congruence rules to extract the call graph from specifications of recursive functions and proves termination of the recursion as specified by the call graph. Such congruence rules describe the *semantic* behaviour of higher-order operators rather than their syntactic specification used for code extraction. Thus, the extracted code for functions may fail to terminate even though termination has been proven by the termination prover. Consider, e.g., the function $\mathsf{ap} : (\mathsf{nat} \to \mathsf{nat}) \to \mathsf{nat} \to \mathsf{nat}$ given by $\mathsf{ap}\ f\ n = \mathsf{if}\ f\ (n+1)\ = 0\ \mathsf{then}\ f\ n\ \mathsf{else}\ f\ n$. The result of $\mathsf{ap}\ f\ n$ only depends on $f\ n$, because both branches of the conditional are the same. This information can be passed to `function` using the custom congruence rule $n = m \longrightarrow f\ m = g\ m \longrightarrow \mathsf{ap}\ f\ n = \mathsf{ap}\ g\ m$. Consequently, the termination prover accepts the recursive specification $\mathsf{foo}\ n = \mathsf{if}\ n = 0\ \mathsf{then}\ 0\ \mathsf{else}\ \mathsf{ap}\ \mathsf{foo}\ (n-1)$ as terminating. Yet, the extracted code for foo does not terminate, as it executes the (pointless) conditional in ap.

induction and coinduction. An AmiCoq would catch up on the corecursion definition front, going beyond what is possible with the *cofix* tactic [20]. On the proof front, AmiCoq would provide a substantial entry into Paco's knowledge base: For any codatatype J with destructor dtor : $\mathsf{J} \to \mathsf{J}\ \mathsf{K}$, all registered friends are, in Paco's terminology, respectful up-to functions for the monotonic operator $\lambda r\ x\ y.\ \mathsf{rel}_{\mathsf{K}}\ r\ (\mathsf{dtor}\ x)\ (\mathsf{dtor}\ y)$, whose greatest fixpoint is the equality on J.

A more lightweight application of our methodology would be an AmiCo for Haskell or for more specialized languages such as CoCaml [41]. In these languages, parametricity is ensured by the computational model. An automatic tool that embodies AmiCo's principles could analyze a Haskell program and prove it total. For CoCaml, which is total, a tool could offer more flexibility when writing corecursive programs.

Surface Synthesis beyond Corecursion The notion of extracting a parametric component with suitable properties can be useful in other contexts than corecursion. In the programming-by-examples paradigm [31], one needs to choose between several synthesized programs whose behavior matches a set of input–output instances. These criteria tend to prefer programs that are highly parametric. A notion of degree of parametricity does not exist in the literature but could be expressed as the size of a parametric surface, for a suitable notion of surface, where $\langle \mathsf{id}, \mathsf{dtor} \rangle$ is replaced by domain specific functions and fst by their left inverses.

Acknowledgment Martin Desharnais spent months extending Isabelle's `codatatype` command to generate a wealth of theorems, many of which were useful when implementing AmiCo. Lorenz Panny developed `primcorec`, whose code provided valuable building blocks. Mathias Fleury, Mark Summerfield, Daniel Wand, and the anonymous reviewers suggested many textual improvements. We thank them all. Blanchette is supported by the European Research Council (ERC) starting grant Matryoshka (713999). Lochbihler is supported by the Swiss National Science Foundation (SNSF) grant "Formalising Computational Soundness for Protocol Implementations" (153217). Popescu is supported by the UK Engineering and Physical Sciences Research Council (EPSRC) starting grant "VOWS: Verification of Web-based Systems" (EP/N019547/1). The authors are listed in alphabetical order.

References

1. Abbott, M., Altenkirch, T., Ghani, N.: Containers: Constructing strictly positive types. Theor. Comput. Sci. 342(1), 3–27 (2005)
2. Abel, A.: MiniAgda: Integrating sized and dependent types. In: Bove, A., Komendantskaya, E., Niqui, M. (eds.) PAR 2010. EPTCS, vol. 43, pp. 14–28 (2010)
3. Abel, A.: Compositional coinduction with sized types. In: Hasuo, I. (ed.) CMCS 2016. LNCS, vol. 9608, pp. 5–10. Springer (2016)
4. Abel, A., Pientka, B.: Well-founded recursion with copatterns and sized types. J. Funct. Program. 26, e2 (2016)
5. Abel, A., Pientka, B., Thibodeau, D., Setzer, A.: Copatterns: Programming infinite structures by observations. In: Giacobazzi, R., Cousot, R. (eds.) POPL 2013. pp. 27–38. ACM (2013)
6. Adams, M.: Introducing HOL Zero (extended abstract). In: Fukuda, K., van der Hoeven, J., Joswig, M., Takayama, N. (eds.) ICMS 2010. LNCS, vol. 6327, pp. 142–143. Springer (2010)

7. Asperti, A., Ricciotti, W., Coen, C.S., Tassi, E.: The Matita interactive theorem prover. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE-23. LNCS, vol. 6803, pp. 64–69. Springer (2011)
8. Atkey, R., McBride, C.: Productive coprogramming with guarded recursion. In: Morrisett, G., Uustalu, T. (eds.) ICFP 2013. pp. 197–208. ACM (2013)
9. Berger, U., Schwichtenberg, H.: An inverse of the evaluation functional for typed lambda-calculus. In: LICS '91. pp. 203–211. IEEE Computer Society (1991)
10. Bernardy, J.P., Jansson, P., Paterson, R.: Proofs for free: Parametricity for dependent types. J. Funct. Program. 22(2), 107–152 (2012)
11. Bertot, Y.: Filters on coinductive streams, an application to Eratosthenes' sieve. In: Urzyczyn, P. (ed.) TLCA 2005. LNCS, vol. 3461, pp. 102–115. Springer (2005)
12. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science, Springer (2004)
13. Bertot, Y., Komendantskaya, E.: Inductive and coinductive components of corecursive functions in Coq. Electr. Notes Theor. Comput. Sci. 203(5), 25–47 (2008)
14. Blanchette, J.C., Bouzy, A., Lochbihler, A., Popescu, A., Traytel, D.: Archive associated with this paper. `http://matryoshka.gforge.inria.fr/pubs/amico_material.tar.gz`
15. Blanchette, J.C., Hölzl, J., Lochbihler, A., Panny, L., Popescu, A., Traytel, D.: Truly modular (co)datatypes for Isabelle/HOL. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 93–110. Springer (2014)
16. Blanchette, J.C., Popescu, A., Traytel, D.: Foundational extensible corecursion: A proof assistant perspective. In: Fisher, K., Reppy, J.H. (eds.) ICFP 2015. pp. 192–204. ACM (2015)
17. Blanchette, J.C., Popescu, A., Traytel, D.: Witnessing (co)datatypes. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 359–382. Springer (2015)
18. Bove, A., Dybjer, P., Norell, U.: A brief overview of Agda—A functional language with dependent types. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 73–78. Springer (2009)
19. Bulwahn, L., Krauss, A., Nipkow, T.: Finding lexicographic orders for termination proofs in Isabelle/HOL. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 38–53. Springer (2007)
20. Chlipala, A.: Certified Programming with Dependent Types—A Pragmatic Introduction to the Coq Proof Assistant. MIT Press (2013)
21. Clouston, R., Bizjak, A., Grathwohl, H.B., Birkedal, L.: Programming and reasoning with guarded recursion for coinductive types. In: Pitts, A.M. (ed.) FoSSaCS 2015. LNCS, vol. 9034, pp. 407–421. Springer (2015)
22. Danielsson, N.A.: Beating the productivity checker using embedded languages. In: Bove, A., Komendantskaya, E., Niqui, M. (eds.) PAR 2010. EPTCS, vol. 43, pp. 29–48 (2010)
23. Dénès, M.: [Coq-Club] Propositional extensionality is inconsistent in Coq (2013), archived at `https://sympa.inria.fr/sympa/arc/coq-club/2013-12/msg00119.html`
24. Devillers, M., Griffioen, D., Müller, O.: Possibly infinite sequences in theorem provers: A comparative study. In: Gunter, E.L., Felty, A. (eds.) TPHOLs '97. LNCS, vol. 1275, pp. 89–104. Springer (1997)
25. Dijkstra, E.W.: An exercise for Dr. R. M. Burstall. In: Selected Writings on Computing: A Personal Perspective, pp. 215–216. Texts and Monographs in Computer Science, Springer (1982)
26. Erwig, M., Kollmansberger, S.: Probabilistic functional programming in Haskell. J. Funct. Programm. 16(1), 21–34 (2006)
27. Gammie, P., Lochbihler, A.: The Stern–Brocot tree. Archive of Formal Proofs (2015), `https://www.isa-afp.org/entries/Stern_Brocot.shtml`

28. Giménez, E.: Codifying guarded definitions with recursive schemes. In: Dybjer, P., Nordström, B., Smith, J.M. (eds.) TYPES '94. LNCS, vol. 996, pp. 39–59. Springer (1995)
29. Giménez, E.: An application of co-inductive types in Coq: Verification of the alternating bit protocol. In: Berardi, S., Coppo, M. (eds.) TYPES '95. LNCS, vol. 1158, pp. 135–152. Springer (1996)
30. Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. Cambridge University Press (1993)
31. Gulwani, S.: Programming by examples—and its applications in data wrangling. In: Dependable Software Systems Engineering, NATO Science for Peace and Security Series—D: Information and Communication Security, vol. 45, pp. 137–158. IOS Press (2016)
32. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 103–117. Springer (2010)
33. Hagino, T.: A Categorical Programming Language. Ph.D. thesis, University of Edinburgh (1987)
34. Harrison, J.: Inductive definitions: Automation and application. In: Schubert, E.T., Windley, P.J., Alves-Foss, J. (eds.) TPHOLs '95. LNCS, vol. 971, pp. 200–213. Springer (1995)
35. Harrison, J.: HOL Light: An overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 60–66. Springer (2009)
36. Hinze, R.: The Bird tree. J. Func. Programm. 19(5), 491–508 (2009)
37. Homeier, P.V.: The HOL-Omega logic. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 244–259. Springer (2009)
38. Huffman, B.: HOLCF '11: A Definitional Domain Theory for Verifying Functional Programs. Ph.D. thesis, Portland State University (2012)
39. Huffman, B., Kunčar, O.: Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In: Gonthier, G., Norrish, M. (eds.) CPP 2013. LNCS, vol. 8307, pp. 131–146. Springer (2013)
40. Hur, C.K., Neis, G., Dreyer, D., Vafeiadis, V.: The power of parameterization in coinductive proof. In: Giacobazzi, R., Cousot, R. (eds.) POPL 2013. pp. 193–206. ACM (2013)
41. Jeannin, J., Kozen, D., Silva, A.: Language constructs for non-well-founded computation. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 61–80. Springer (2013)
42. Jones, G., Gibbons, J.: Linear-time breadth-first tree algorithms: An exercise in the arithmetic of folds and zips. Tech. Rep. 71, C.S. Dept., University of Auckland (1993)
43. Keller, C., Lasson, M.: Parametricity in an impredicative sort. In: Cégielski, P., Durand, A. (eds.) CSL 2012. LIPIcs, vol. 16, pp. 381–395. Schloss Dagstuhl—Leibniz-Zentrum für Informatik (2012)
44. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. 6(2), 323–350 (1977)
45. Krauss, A.: Partial recursive functions in higher-order logic. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS, vol. 4130, pp. 589–603. Springer (2006)
46. Krauss, A.: Recursive definitions of monadic functions. In: Bove, A., Komendantskaya, E., Niqui, M. (eds.) PAR 2010. EPTCS, vol. 43, pp. 1–13 (2010)
47. van Laarhoven, T.: Knuth–Morris–Pratt in Haskell. `http://www.twanvl.nl/blog/haskell/Knuth-Morris-Pratt-in-Haskell` (2007)
48. Leino, K.R.M.: Automating theorem proving with SMT. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 2–16. Springer (2013)
49. Leino, K.R.M., Moskal, M.: Co-induction simply: Automatic co-inductive proofs in a program verifier. In: Jones, C.B., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 382–398. Springer (2014)
50. Lochbihler, A.: Probabilistic functions and cryptographic oracles in higher-order logic. In: Thiemann, P. (ed.) ESOP 2016. LNCS, vol. 9632, pp. 503–531. Springer (2016)

51. Lochbihler, A., Hölzl, J.: Recursive functions on lazy lists via domains and topologies. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 341–357. Springer (2014)
52. Lochbihler, A., Züst, M.: Programming TLS in Isabelle/HOL. Isabelle Workshop 2014 (2014), `https://www.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/information-security-group-dam/research/publications/pub2014/lochbihler14iw.pdf`
53. Lucanu, D., Goriac, E.I., Caltais, G., Roşu, G.: CIRC: A behavioral verification tool based on circular coinduction. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) CALCO 2009. LNCS, vol. 5728, pp. 433–442. Springer (2009)
54. Matthews, J.: Recursive function definition over coinductive types. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs '99. LNCS, vol. 1690, pp. 73–90. Springer (1999)
55. Milius, S.: Completely iterative algebras and completely iterative monads. Inf. Comput. 196(1), 1–41 (2005)
56. Milius, S., Moss, L.S., Schwencke, D.: Abstract GSOS rules and a modular treatment of recursive definitions. Log. Meth. Comput. Sci. 9(3) (2013)
57. Moss, L.S.: Parametric corecursion. Theor. Comput. Sci. 260(1-2), 139–163 (2001)
58. de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The Lean theorem prover (system description). In: Felty, A.P., Middeldorp, A. (eds.) CADE-25. LNCS, vol. 9195, pp. 378–388. Springer (2015)
59. Myreen, M.O.: Functional programs: Conversions between deep and shallow embeddings. In: Beringer, L., Felty, A. (eds.) ITP 2012. pp. 412–417. LNCS, Springer (2012)
60. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer (2002)
61. Paulson, L.C.: A fixedpoint approach to implementing (co)inductive definitions. In: Bundy, A. (ed.) CADE-12. LNCS, vol. 814, pp. 148–161. Springer (1994)
62. Paulson, L.C.: Mechanizing coinduction and corecursion in higher-order logic. J. Log. Comput. 7(2), 175–204 (1997)
63. Pous, D.: Coinduction all the way up. In: Grohe, M., Koskinen, E., Shankar, N. (eds.) LICS 2016. pp. 307–316. ACM (2016)
64. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: Mason, R.E.A. (ed.) IFIP '83. pp. 513–523. North-Holland/IFIP (1983)
65. Rot, J., Bonsangue, M.M., Rutten, J.J.M.M.: Coalgebraic bisimulation-up-to. In: van Emde Boas, P., Groen, F.C.A., Italiano, G.F., Nawrocki, J.R., Sack, H. (eds.) SOFSEM 2013. LNCS, vol. 7741, pp. 369–381. Springer (2013)
66. Rutten, J.J.M.M.: Universal coalgebra: A theory of systems. Theor. Comput. Sci. 249, 3–80 (2000)
67. Rutten, J.J.M.M.: Automata and coinduction (an exercise in coalgebra). In: Sangiorgi, D., de Simone, R. (eds.) CONCUR '98. LNCS, vol. 1466, pp. 194–218. Springer (1998)
68. Slind, K.: Function definition in higher-order logic. In: von Wright, J., Grundy, J., Harrison, J. (eds.) TPHOLs '96. LNCS, vol. 1125, pp. 381–397. Springer (1996)
69. Slind, K., Norrish, M.: A brief overview of HOL4. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 28–32. Springer (2008)
70. Thibodeau, D., Cave, A., Pientka, B.: Indexed codata types. In: Sumii, E. (ed.) ICFP 2016. ACM (2016)
71. Traytel, D.: [Agda] Agda's copatterns incompatible with initial algebras (2014), archived at `https://lists.chalmers.se/pipermail/agda/2014/006759.html`
72. Traytel, D.: Formal languages, formally and coinductively. In: Kesner, D., Pientka, B. (eds.) FSCD 2016. LIPIcs, vol. 52, pp. 31:1–17. Schloss Dagstuhl—Leibniz-Zentrum für Informatik (2016)

73. Traytel, D., Popescu, A., Blanchette, J.C.: Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In: LICS 2012, pp. 596–605. IEEE Computer Society (2012)
74. Turner, D.A.: Elementary strong functional programming. In: Hartel, P.H., Plasmeijer, M.J. (eds.) FPLE '95. LNCS, vol. 1022, pp. 1–13. Springer (1995)
75. Wadler, P.: Theorems for free! In: Stoy, J.E. (ed.) FPCA 1989. pp. 347–359. ACM (1989)
76. Winskel, G.: A note on model checking the modal ν-calculus. Theor. Comput. Sci. 83(1), 157–167 (1991)