Microsoft Research Technical Report MSR-TR-2013-49

Co-induction Simply

Automatic Co-inductive Proofs in a Program Verifier

K. Rustan M. Leino Michał Moskal

Microsoft Research, Redmond, WA, USA
{leino,micmo}@microsoft.com

Abstract

Program verification relies heavily on induction, which has received decades of attention in mechanical verification tools. When program correctness is best described by infinite structures, program verification is usefully aided also by co-induction, which has not benefited from the same degree of tool support. Co-induction is complicated to work with in interactive proof assistants and has had no previous support in dedicated program verifiers.

This paper shows that an SMT-based program verifier can support reasoning about co-induction—handling infinite data structures, lazy function calls, and user-defined properties defined as greatest fix-points, as well as letting users write co-inductive proofs. Moreover, the support can be packaged to provide a simple user experience. The paper describes the features for co-induction in the language and verifier Dafny, defines their translation into input for a first-order SMT solver, and reports on some encouraging initial experience.

Categories and Subject Descriptors D.2.4 [*SOFTWARE ENGINEERING*]: Software/Program Verification

General Terms Verification, Co-induction.

Keywords verification, SMT, co-induction, lazy data structures

0. Introduction

Mathematical induction is a cornerstone of programming and program verification. It arises in data definitions (*e.g.*, some algebraic data structures can be described using induction [4]), it underlies program semantics (*e.g.*, it explains how to reason about finite iteration and recursion [1]), and it gets used in proofs (*e.g.*, supporting lemmas about data structures use inductive proofs [18]). Whereas induction deals with finite things (data, behavior, etc.), its dual, *co-induction*, deals with possibly infinite things. Co-induction, too, is important in programming and program verification, where it arises in data definitions (*e.g.*, lazy data structures [32]), semantics (*e.g.*, concurrency [30]), and proofs (*e.g.*, showing refinement in a co-inductive big-step semantics [23]). It is thus desirable to have good support for both induction and co-induction in a system for constructing and reasoning about programs.

Dramatic improvements in satisfiability-modulo-theories (SMT) solvers have brought about new levels of power in automated reasoning. Some program verifiers and interactive proof assistants have used this power to reduce the amount of human interaction needed to achieve results (*e.g.*, [5, 9, 14, 20]). In this paper, we introduce the first SMT-based verifier to support co-induction.

The verifier is for programs written in the verification-aware programming language Dafny [20],[0] which we extend with co-inductive features. Co-datatypes and co-recursive functions make it possible to use lazily evaluated data structures (like in Haskell [32] or Agda [28]). Co-predicates, defined by greatest fix-points, let programs state properties of such data structures (as can also be done in, for example, Coq [3]). For the purpose of writing co-inductive proofs in the language, we introduce *co-methods*. Ostensibly, a co-method invokes the co-induction hypothesis much like an inductive proof invokes the induction hypothesis. Underneath the hood, our co-inductive proofs are actually approached via induction [25]: co-methods provide a syntactic veneer around this approach.

These language features and the automation in our SMT-based verifier combine to provide a simple view of co-induction. As a sneak peek, consider the program in Fig. 0. It defines a type `IStream` of infinite streams, with constructor `ICons` and destructors `head` and `tail`. Function `Mult` performs pointwise multiplication on infinite streams of integers, defined using a co-recursive call (which is evaluated lazily). Co-predicate `Below` is defined as a greatest fix-point, which intuitively means that the co-predicate will take on the value **`true`** if the recursion goes on forever without determining a different value. The co-method states the theorem `Below(a, Mult(a, a))`. Its body gives the proof, where the recursive invocation of the co-method corresponds to an invocation of the co-induction hypothesis.

We argue that these definitions in Dafny are simple enough to level the playing field between induction (which is familiar) and co-induction (which, despite being the dual of induction, is often perceived as eerily mysterious). Moreover, the automation provided by our SMT-based verifier reduces the tedium in writing co-inductive proofs. For example, it verifies `Theorem_BelowSquare` from the program text given in Fig. 0—no additional lemmas or tactics are needed. In fact, as a consequence of the automatic-induction heuristic in Dafny [21], the verifier will automatically verify `Theorem_BelowSquare` even given an empty body.

Just like there are restrictions on when an *inductive* hypothesis can be *invoked*, there are restriction on how a *co-inductive* hypothesis can be *used*. These are, of course, taken into consideration by our verifier. For example, as illustrated by the second co-method in Fig. 0, invoking the co-inductive hypothesis in an attempt to obtain the entire proof goal is futile. (We explain how this works in Sect. 3.1.)

Until recently, Dafny has primarily been a verifier for imperative programs. The introduction of inductive features, and now our co-

[0] Dafny is an open-source project at `http://dafny.codeplex.com`.

```
// infinite streams
codatatype IStream<T> = ICons(head: T, tail: IStream)
// pointwise product of streams
function Mult(a: IStream<int>, b: IStream<int>): IStream<int>
{
  ICons(a.head * b.head, Mult(a.tail, b.tail))
}
// lexicographic order on streams
copredicate Below(a: IStream<int>, b: IStream<int>)
{
  a.head ≤ b.head ∧
  (a.head = b.head ⟹ Below(a.tail, b.tail))
}
// a stream a Below its Square
comethod Theorem_BelowSquare(a: IStream<int>)
  ensures Below(a, Mult(a, a));
{
  assert a.head ≤ Mult(a, a).head;
  if a.head = Mult(a, a).head {
    Theorem_BelowSquare(a.tail);
  }
}
// an incorrect property and a bogus proof attempt
comethod NotATheorem_SquareBelow(a: IStream<int>)
  ensures Below(Mult(a, a), a);  // ERROR
{
  NotATheorem_SquareBelow(a);
}
```

Figure 0. A taste of how the co-inductive features in Dafny come together to give straightforward definitions of infinite matters. The proof of the theorem stated by the first co-method lends itself to the following intuitive reading: To prove that a is below Mult(a, a), check that their heads are ordered and, if the heads are equal, also prove that the tails are ordered. The second co-method states a property that does not always hold; the verifier is not fooled by the bogus proof attempt and instead reports the property as unproved. UkWw4 »

inductive extension, give a boost to the functional and theorem-proving capabilities of the system. The imperative and functional constructs work well together, especially when using functions in the specifications of classes and methods.

Our initial experience with co-induction in Dafny shows it to provide an intuitive, low-overhead user experience that compares favorably to even the best of today's interactive proof assistants for co-induction. In addition, the co-inductive features and verification support in Dafny have other potential benefits. The features are a stepping stone for verifying functional lazy programs with Dafny. Co-inductive features have also shown to be useful in defining language semantics, as needed to verify the correctness of a compiler [23], so this opens the possibility that such verifications can benefit from SMT automation.

0.0 Contributions

- First SMT-based verifier for reasoning about co-induction.
- Language design that blends inductive and co-inductive features, allowing both recursive and *co-recursive calls* to the same function (Sect. 2).
- User-callable *prefix predicates*—finite unfoldings of co-predicates used to establish co-predicates via induction (Sects. 2.2 and 5).
- Extension of the technique of writing inductive proofs as programs (see background in Sect. 1) to co-inductive proofs using *co-methods* (Sect. 3). Unlike tactic-based systems, these programs show the high-level structure of the (inductive and co-inductive) proofs. Yet the automation provided by the SMT solver makes it unnecessary to manually author the proof terms.
- Low-overhead tool-supported way to write and learn about co-inductive proofs (see examples in Sect. 4).

1. Background: Proofs and Induction

In this section, we review the use of mathematical inductive proofs in the program verifier Dafny. This will establish a foundation for the discussion of co-induction in the subsequent sections.

1.0 Functions

Dafny programs are sets of declarations (see Fig. 1). The Dafny programming language supports functions and methods. A *function* in Dafny is a mathematical function (*i.e.*, it is well-defined, deterministic, and pure), whereas a *method* is a body of statements that can mutate the state of the program. A function is defined by its given body, which is an expression. Figure 2 shows a function computing element n of the Fibonacci sequence.[1]

To ensure that function definitions are mathematically consistent, Dafny insists that recursive calls be well-founded, enforced as follows: Dafny computes the call graph of functions. The strongly connected components within it are *clusters* of mutually recursive definitions arranged in a DAG. This stratifies the functions so that a call from one cluster in the DAG to a lower cluster is allowed arbitrarily. For an intra-cluster call, Dafny prescribes a proof obligation that gets taken through the program verifier's reasoning engine. Semantically, each function activation is labeled by a *rank*—a lexicographic tuple determined by evaluating the function's **decreases** clause upon invocation of the function. The proof obligation for an intra-cluster call is thus that the rank of the callee is strictly less (in a language-defined well-founded relation) than the rank of the caller [20]. For example, for the Fib function, if the rank of the caller is the natural number n, then the callees have ranks $n - 2$ and $n - 1$, so Dafny's well-founded checks pass. Because these well-founded checks correspond to proving termination of executable code, we will often refer to them as "termination checks". The same process applies to methods.

Dafny uses a simple heuristic that, in most common cases, avoids the need for explicit **decreases** declarations. However, in this paper, we show all **decreases** clauses explicitly, which also serves as a reminder about which termination checks apply.

1.1 Lemmas

When proving the full correctness of a program, verification often requires certain properties of user-defined functions. These can be stated and proved as *lemmas*. Being a programming language, Dafny does not have any **lemma** or **proof** declaration. Instead, a lemma is commonly introduced by declaring a method, stating the property of the lemma in the *postcondition* (keyword **ensures**) of the method, perhaps restricting the domain of the lemma by also giving a *precondition* (keyword **requires**), and using the lemma by invoking the method [16, 21]. For example, method FibLemma in Fig. 2 states the property that Fib(n) is even exactly when n is a multiple of 3, and the two recursive calls are examples of uses of the lemma.

When a lemma is given as a method postcondition, the proof of the lemma consists in verifying that all execution paths of the method body establish the postcondition—this is nothing but the

[1] The examples in the figures can be tried and tweaked online at the following address http://rise4fun.com/Dafny/id where id is provided below every figure.

```
program  ::= decl*

decl ::=
    function        sig ':' typ decr '{' expr '}'
  | function method sig ':' typ decr '{' expr '}'
  | predicate       sig         decr '{' expr '}'
  | copredicate     sig              '{' expr '}'
  | method          methdef
  | ghost method    methdef
  | comethod        methdef
  | datatype        id typargs '=' sig ('|' sig)*
  | codatatype      id typargs '=' sig ('|' sig)*

sig      ::= id typargs '(' [ formal, ... ] ')'
formal   ::= id ':' typ
typ      ::= nat | int | bool | id [ '<' typ, ... '>' ]
typargs  ::= [ '<' id, ... '>' ]
methdef  ::= sig contract* [decr] block
contract ::= requires expr | ensures expr
decr     ::= decreases expr

stmt ::=
    id '(' [ expr, ... ] ')'
  | assert expr
  | if expr block [ else block ]
  | forall id, ... '|' expr block
  | match expr '{' (case pat ⇒ stmt*)* '}'
block ::= '{' stmt* '}'

expr ::=
    false | true | 0 | 1 | ...
  | id
  | id '(' [ expr, ... ] ')'
  | expr '.' id
  | expr op expr | ¬ expr
  | ∀ id, ... • expr
  | ∃ id, ... • expr
  | if expr then expr else expr
  | match expr (case pat ⇒ expr)*

pat ::= id [ '(' id, ... ')' ]
op  ::= ∧ | ∨ | ⟹ | ⟺ | + | - | * | / | %
      | = | ≠ | ≤ | < | > | ≥
```

Figure 1. Syntax of the subset of Dafny used in this paper. `t*` denotes `t` repeated 0 or more times, `[t]` denotes `t` repeated 0 or 1 times, `t, ...` denotes `t (',' t)*`, and `()` are used for grouping. We ignore semicolons and operator precedence here. Dafny also includes partial functions, a wide range of imperative constructs, class declarations, and a module system supporting refinement, but these are not used in this paper.

```
function Fib(n: nat): nat
  decreases n;
{ if n < 2 then n else Fib(n-2) + Fib(n-1) }

ghost method FibLemma(n: nat)
  ensures Fib(n) % 2 = 0 ⟺ n % 3 = 0;
  decreases n;
{
  if n < 2 {
  } else {
    FibLemma(n-2);
    FibLemma(n-1);
  }
}
```

Figure 2. The well-known Fibonacci function defined in Dafny. The postcondition of method `FibLemma` states a property about `Fib`, and the body of the method is code that convinces the program verifier that the postcondition does indeed hold. Thus, effectively, the method states a lemma and its body gives an inductive proof. `jHRq` »

ordinary treatment of postconditions in the context of program verification. One way of writing the `FibLemma` proof is shown in Fig. 2, where the method body splits the execution paths based on whether or not $n < 2$ holds. In mathematics, we say that the proof does a *case split*. For the $n < 2$ case, the program verifier is able to construct the proof automatically, simply by applying the definition of `Fib`. In the other case, the body makes two recursive calls. The program verifier then, in its usual manner, checks for each call that the precondition holds (which, here, simply involves checking that the integer passed as an argument is a natural number) and checks that the recursive calls terminate (that is, comparing the ranks of the caller and callee according to the **decreases** clause).[2] Upon return of each call, the program verifier gets to assume—again, as a matter of course in program verification—that the postcondition, instantiated with the actual parameters, holds. In mathematics, this corresponds to invoking the *induction hypothesis*. The remaining proof glue, to go from the two induction hypotheses to the proof goal, is done automatically by the tool.

Note that there is no explicit proof object being created. Instead, the underlying SMT solver is asked to prove the postcondition. This allows user-supplied proof steps to be bigger, more akin to using exploratory tactics than explicit proofs in an interactive proof assistant.

It is important to note that the program text shown in Fig. 2 (and, indeed, all the examples in this paper) is all that is given to the program verifier. That is, there is no further need to guide the prover in how to carry out the proof.[3]

Lemmas are stated, used, and proved as methods, which is useful when the program is verified, but there is no reason to have such methods available at run time. Indeed, execution of the method `FibLemma` would do nothing, except consume time and stack space. For this reason, a lemma method is typically declared as *ghost*, meaning that it is not compiled into code. The concept of ghost versus *non-ghost* (that is, "to be compiled" or "physical") declarations is an integral part of the Dafny language: each function, method, variable, and parameter can be declared as either ghost or non-ghost. Functions are by default ghost, and to make them executable one needs to say **function method**. Non-ghost code is subject to certain restrictions, *e.g.*, one can only use quantifiers over finite ranges.

[2] The verifier also needs to check that the state mutations performed by the callee are ones the caller is allowed to perform. In this paper, we are not concerned with mutations, so we will not dwell on this point.

[3] If the verifier is unable to complete a proof, it will generate an error message and it is then up to the user to figure out how to provide more hints. This is analogous to what happens when a tactic in an interactive proof assistant fails. How such debugging is done lies outside the scope of this paper, but we invite interested readers to learn about the Boogie Verification Debugger [19], which lets the programmer inspect the counterexample (or failed proof state, if you will) using an interface similar to a regular debugger.

```
ghost method FibLemma(n: nat)
  ensures Fib(n) % 2 = 0 ⟺ n % 3 = 0;
{
  forall k | 0 ≤ k < n {
    FibLemma(k);
  }
}

ghost method FibLemma_All()
  ensures ∀ n • 0 ≤ n ⟹
               (Fib(n) % 2 = 0 ⟺ n % 3 = 0);
{
  forall n | 0 ≤ n {
    FibLemma(n);
  }
}
```

Figure 3. The **forall** statement has the effect of applying its body simultaneously to all values of the bound variables—in the first example, to all k satisfying $0 \leqslant k < n$, and in the second example, to all non-negative n. Function Fib is defined in Fig. 2, which gave a different proof of FibLemma. gY1r »

1.2 Aggregate Operations

Dafny provides a program statement that can perform an aggregate operation: the **forall** statement. It has uses both in ordinary programming (like initializing all elements of an array) and for proofs. We show two example uses of the **forall** statement in Fig. 3.

In the first example, the Fib property stated by FibLemma in Fig. 2 is proved in an alternate way. Instead of invoking the induction hypothesis zero times when $n < 2$ and two times otherwise, this alternative FibLemma makes one recursive call for every k less than n. If this code were executed, it would make an enormous total number of recursive calls, but the method is declared as ghost, so no worries. The use of the induction hypothesis for all values smaller than in the proof goal is in mathematics referred to as *strong induction*. (In this particular example, only two of those induction hypotheses are actually useful.)

Dafny sets up some proofs by induction automatically, according to a simple heuristic [21]. In the case of FibLemma, Dafny will in fact insert the strong-induction **forall** statement automatically, so the program verifier successfully carries out the proof even if the user left the body of this ghost method empty.

The second example in Fig. 3 shows what in mathematics is called *universal introduction*. Lemma FibLemma is parameterized by any n and states a property of the form $P(n)$. According to the rule of universal introduction, what FibLemma states for an arbitrary n, it actually states universally for all n. The lemma FibLemma_All states the universal property, $\forall n \bullet P(n)$. The proof of FibLemma_All uses the **forall** statement to invoke FibLemma for every value of n. Note, the fact that the range of n is infinite is not a worry, since this is just ghost code anyway.

1.3 Inductive Datatypes

Some data structures are conveniently defined inductively, like the common *inductive datatype* List in Fig. 4. The datatype List is parameterized by a type T and has two constructors, Nil (for constructing an empty list) and Cons (for constructing a nonempty list from an element and another list). In Dafny, each constructor C automatically gives rise to a discriminator C?, and each parameter of a constructor can be named in order to introduce a corresponding destructor. For example, if xs is the list Cons(x, ys), then xs.Cons? and xs.head = x hold.

```
datatype List<T> = Nil | Cons(head: T, tail: List)

function Append(xs: List, ys: List): List
  decreases xs;
{
  match xs
  case Nil ⇒ ys
  case Cons(x, rest) ⇒ Cons(x, Append(rest, ys))
}

ghost method AppendIsAssociative(xs: List, ys: List,
                                 zs: List)
  ensures Append(Append(xs, ys), zs) =
          Append(xs, Append(ys, zs));
  decreases xs;
{
  match xs {
    case Nil ⇒
    case Cons(x, rest) ⇒
      AppendIsAssociative(rest, ys, zs);
  }
}
```

Figure 4. A standard inductive definition of a generic List type and a function Append that concatenates two lists. The ghost method states the lemma that Append is associative, and its body gives the inductive proof. In several places, we omit the explicit declarations and uses of List's type parameter, since in common cases, Dafny is able to fill these in automatically. ntQL »

Dafny insists that every type be inhabited, which is ensured by a syntactic check of *grounding* in each cluster of mutually recursive datatypes. For example, if the Nil constructor were omitted from Fig. 4, then Dafny would complain that cyclic dependencies prevent the construction of any List value.

A value of an inductive datatype is a finite tree, where each internal node is a constructor of some inductive datatype and each leaf is a nullary constructor of some inductive datatype, a pointer into the heap, or a value of some co-inductive datatype. For this purpose, booleans and integers are inductive datatypes (represented during reasoning as SMT integers and at run time as big-nums). Dafny predefines a partial well-founded order ("proper subtree of") on values of inductive datatypes. Pointers and values of co-inductive datatypes are not included in that order.

For inductive datatypes, it is natural to give related functions and proofs inductively as well. For example, Figure 4 defines a function Append on lists and proves a lemma that Append is associative. The rank of the function (and method, respectively), as given by the **decreases** clause, is the datatype value xs. The body of the function uses a typical **match** expression, which gives a convenient way to deconstruct values. The method uses an analogous **match** statement. As in the FibLemma example, the body of AppendIsAssociative can be left empty—the implicit **forall** statement inserted by Dafny's induction tactic is enough to convince the verifier that the postcondition holds.

2. Co-inductive Definitions

In this section and the next, we describe the design of our co-inductive extension of Dafny. We start with the constructs for defining types, values, and properties of possibly infinite data structures.

```
codatatype Stream<T> = SNil | SCons(head: T, tail: Stream)

function Up(n: int): Stream<int>
{ SCons(n, Up(n+1)) }

function FivesUp(n: int): Stream<int>
  decreases 4 - (n - 1) % 5;
{ if n % 5 = 0 then
  SCons(n, FivesUp(n+1)) else FivesUp(n+1) }
```

Figure 5. Stream is a co-inductive datatype, whose values are possibly infinite lists. Function Up returns a stream consisting of all integers upwards of n and FivesUp returns a stream consisting of all multiples of 5 upwards of n. The self-call in Up and the first self-call in FivesUp sit in productive positions and are therefore classified as co-recursive calls, exempt from termination checks. The second self-call in FivesUp is not in a productive position and is therefore subject to termination checking; in particular, each recursive call must decrease the rank defined by the **decreases** clause. CplhV »

2.0 Defining Co-inductive Datatypes

Each value of an inductive datatype is finite, in the sense that it can be constructed by a finite number of calls to datatype constructors. In contrast, values of a *co-inductive datatype*, or *co-datatype* for short, can be infinite. For example, a co-datatype can be used to represent infinite trees.

Syntactically, the declaration of a co-datatype in Dafny looks like that of a datatype, giving prominence to the constructors (following Coq [11]). Like for inductive datatypes, each constructor automatically gives rise to a discriminator, each parameter of a constructor can be named to introduce a corresponding destructor, and **match** expressions and statements can also be used to destruct values. For example, Fig. 5 defines a co-datatype Stream of possibly infinite lists. Like the analogous finite list datatype, Stream declares two constructors, SNil and SCons, and names the two destructors of SCons streams. In contrast to datatype declarations, there is no grounding check for co-datatypes—since a co-datatype admits infinite values, the type is nevertheless inhabited.

2.1 Creating Values of Co-datatypes

To define values of co-datatypes, one could imagine a "co-function" language feature: the body of a "co-function" could include possibly never-ending self-calls that are interpreted by a greatest fix-point semantics (like a CoFixpoint in Coq). Dafny uses a different design: it offers only functions (not "co-functions"), but it classifies each intra-cluster call as either *recursive* or *co-recursive*. Recursive calls are subject to termination checks, as we described in the previous section. Co-recursive calls may be never-ending, which is what is needed to define infinite values of a co-datatype. For example, function Up(n) in Fig. 5 is defined as the stream of numbers from n upward: it returns a stream that starts with n and continues as the co-recursive call Up($n+1$).

To ensure that co-recursive calls give rise to mathematically consistent definitions, they must occur only in *productive positions*. This says that it must be possible to determine each successive piece of a co-datatype value after a finite amount of work. This condition is satisfied if every co-recursive call is syntactically *guarded* by a constructor of a co-datatype, which is the criterion Dafny uses to classify intra-cluster calls as being either co-recursive or recursive. Calls that are classified as co-recursive are exempt from termination checks.

Because co-recursive calls may be never-ending, it would be no good to eagerly evaluate them at run time. Instead, the Dafny compiler turns co-recursive calls into parameter-less closures, evaluated at run time when the enclosing constructor is destructed (if ever). For example, each co-recursive call to Up in Fig. 5 is evaluated lazily.

```
copredicate Pos(s: Stream<int>)
{
  match s
  case SNil ⇒ true
  case SCons(x, rest) ⇒ x > 0 ∧ Pos(rest)
}
// Automatically generated by the Dafny compiler:
predicate Pos#[_k: nat](s: Stream<int>)
  decreases _k;
{
  if _k = 0 then true else
    match s
    case SNil ⇒ true
    case SCons(x, rest) ⇒ x > 0 ∧ Pos#[_k-1](rest)
}
```

Figure 6. A co-predicate Pos that holds for those integer streams whose every integer is greater than 0. The co-predicate definition implicitly also gives rise to a corresponding prefix predicate, Pos$^{\#}$. The syntax for calling a prefix predicate sets apart the argument that specifies the prefix length, as shown in the last line; for this figure, we took the liberty of making up a coordinating syntax for the signature of the automatically generated prefix predicate. eYml »

A consequence of the productivity checks and termination checks is that, even in the absence of talking about least or greatest fix-points of self-calling functions, all functions in Dafny are deterministic. Since there is no issue of several possible fix-points, the language allows one function to be involved in both recursive and co-recursive calls, as we illustrate by the function FivesUp in Fig. 5.

2.2 Stating Properties of Co-datatypes

Determining properties of co-datatype values may require an infinite number of observations. To that avail, Dafny provides *co-predicates*. Self-calls to a co-predicate need not terminate. Instead, the value defined is the greatest fix-point of the given recurrence equations. Figure 6 defines a co-predicate that holds for exactly those streams whose payload consists solely of positive integers.

Some restrictions apply. To guarantee that the greatest fix-point always exists, the (implicit functor defining the) co-predicate must be monotonic. This is enforced by a syntactic restriction on the form of the body of co-predicates: after conversion to negation normal form (*i.e.*, pushing negations down to the atoms), intra-cluster calls of co-predicates must appear only in *positive* positions—that is, they must appear as atoms and must not be negated. Additionally, to guarantee soundness later on, we require that they appear in *co-friendly* positions—that is, in negation normal form, when they appear under existential quantification, the quantification needs to be limited to a finite range. This is formalized in Fig. 7; in Sect. 4, we also give some examples in Fig. 14. Since the evaluation of a co-predicate might not terminate, co-predicates are always ghost. There is also a restriction on the call graph that a cluster containing a co-predicate must contain only co-predicates, no other kinds of functions.

A **copredicate** declaration P defines not just a co-predicate, but also a corresponding *prefix predicate* P$^{\#}$. A prefix predicate is a finite unrolling of a co-predicate. The prefix predicate is constructed from the co-predicate by

$$
\begin{array}{rcll}
[l]_p & = & \ominus_p\ l & l \text{ is bool or int literal} \\
[x]_p & = & \ominus_p\ x & x \text{ is variable reference} \\
[P(a, \ldots, z)]_1 & = & P([a]_0, \ldots, [z]_0) & \\
[P(a, \ldots, z)]_p & = & \bot & \\
[f(a, \ldots, z)]_p & = & \ominus_p\ f([a]_0, \ldots, [z]_0) & \\
[a.d]_p & = & \ominus_p\ [a]_0.d & \\
[\neg a]_0 & = & \neg [a]_0 & \\
[\neg a]_p & = & [a]_{-p} & \\
[a \wedge b]_{-1} & = & [a]_{-1} \vee [b]_{-1} & \\
[a \vee b]_{-1} & = & [a]_{-1} \wedge [b]_{-1} & \\
[a \wedge b]_p & = & [a]_p \wedge [b]_p & \\
[a \vee b]_p & = & [a]_p \vee [b]_p & \\
[a \implies b]_p & = & [\neg a \vee b]_p & \\
[a \iff b]_p & = & [(a \implies b) \wedge (b \implies a)]_p & \\
[a\ op\ b]_p & = & \ominus_p\ ([a]_0\ op\ [b]_0) & \\
[\exists\, x \bullet\ a]_{-1} & = & [\forall\, x \bullet\ \neg a]_1 & \\
[\forall\, x \bullet\ a]_{-1} & = & [\exists\, x \bullet\ \neg a]_1 & \\
[\forall\, x \bullet\ a]_p & = & \forall\, x \bullet\ [a]_p & \\
[\exists\, x \bullet\ a \le x < b \wedge c]_1 & = & \exists\, x \bullet\ [a \le x < b]_0 \wedge [c]_1 & \\
[\exists\, x \bullet\ a]_p & = & \exists\, x \bullet\ [a]_0 & \\
[\textbf{if}\ a\ \textbf{then}\ b\ \textbf{else}\ c]_p & = & \textbf{if}\ [a]_0\ \textbf{then}\ [b]_p\ \textbf{else}\ [c]_p & \\
[\textbf{match}\ a\ (\textbf{case}\, p \Rightarrow b) \ldots]_p & = & \textbf{match}\ [a]_0\ (\textbf{case}\, p \Rightarrow [b]_p) \ldots &
\end{array}
$$

Figure 7. Rules for co-friendliness. The transformer $[a]_p$ converts expression a into negation normal form, but changes intra-cluster calls to co-predicates not in positive and co-friendly positions into $\bot$. p indicates the context: positive (1), negative (-1), or neither (0). An expression a is allowed as the body of a co-predicate iff $[a]_1$ does not contain a subexpression $\bot$. a, b, c, and z are any expressions, $\ominus_{-1}$ is $\neg$, whereas $\ominus_0$ and $\ominus_1$ are identity, P is an intra-cluster co-predicate invocation, f is an invocation of any function or co-predicate in a lower cluster, d is a destructor, and p is a pattern. Dafny employs a simple, conservative heuristic for detecting finite ranges in existential quantifications. One such heuristic rule is shown above; others apply to the built-in set membership predicate, *etc.*

- adding a parameter _k of type **nat** to denote the *prefix length*,
- adding the clause **decreases** _k; to the prefix predicate (the co-predicate itself is not allowed to have a **decreases** clause),
- replacing in the body of the co-predicate every intra-cluster call Q(*args*) to a co-predicate by a call Q$^{\#}$[_k − 1](*args*) to the corresponding prefix predicate, and then
- prepending the body with **if** _k = 0 **then true else**.

For example, for co-predicate Pos, the definition of the prefix predicate Pos$^{\#}$ is as suggested in Fig. 6. Syntactically, the prefix-length argument passed to a prefix predicate to indicate how many times to unroll the definition is written in square brackets, as in Pos$^{\#}$[k](s). The definition of Pos$^{\#}$ is available only at clusters strictly higher than that of Pos; that is, Pos and Pos$^{\#}$ must not be in the same cluster. In other words, the definition of Pos cannot depend on Pos$^{\#}$.

Equality between two values of a co-datatype is a built-in ghost co-predicate. It has the usual equality syntax $s = t$, and the corresponding prefix equality is written $s \overset{\#}{=}$[k] t.

3. Co-inductive Proofs

From what we have said so far, a program can make use of properties of co-datatypes. For example, a method that declares Pos(s) as a precondition can rely on the stream s containing only positive integers. In this section, we consider how such properties are established in the first place.

```
ghost method UpPosLemmaK(k: nat, n: int)
  requires n > 0;
  ensures Pos#[k](Up(n));
{
  if k ≠ 0 {
    // this establishes Pos#[k-1](Up(n).tail)
    UpPosLemmaK(k-1, n+1);
  }
}

ghost method UpPosLemma(n: int)
  requires n > 0;
  ensures Pos(Up(n));
{
  forall k | 0 ≤ k { UpPosLemmaK(k, n); }
}
```

Figure 8. The method UpPosLemma proves Pos(Up(n)) for every $n > 0$. We first show Pos$^{\#}$[k](Up(n)), for $n > 0$ and an arbitrary k, and then use the **forall** statement to show ∀ k • Pos$^{\#}$[k](Up(n)). Finally, the axiom $\mathcal{D}$(Pos) is used (automatically) to establish the co-predicate. kSAwb »

3.0 Properties About Prefix Predicates

Among other possible strategies for establishing co-inductive properties (*e.g.*, [7, 15]), we take the time-honored approach of reducing co-induction to induction [25]. More precisely, Dafny passes to the SMT solver an assumption $\mathcal{D}(P)$ for every co-predicate P, where:

$$\mathcal{D}(P) \quad \equiv \quad \forall\, \overline{x} \bullet\ P(\overline{x}) \iff \forall\, k \bullet\ P^{\# k}(\overline{x})$$

In Sect. 5, we prove soundness of such assumptions, provided the co-predicates meet the co-friendly restrictions from Sect. 2.2. An example proof of Pos(Up(n)) for every $n > 0$ is shown in Fig. 8.

3.1 Co-methods

As we just showed, with help of the $\mathcal{D}$ axiom we can now prove a co-predicate by inductively proving that the corresponding prefix predicate holds for all prefix lengths k. In this section, we introduce *co-method* declarations, which bring about two benefits. The first benefit is that co-methods are syntactic sugar and reduce the tedium of having to write explicit quantifications over k. The second benefit is that, in simple cases, the bodies of co-methods can be understood as co-inductive proofs directly.

We start by illustrating the second benefit. Figure 9 shows the definition of an append function for streams and a proof, using a co-method, that append is associative. Note that the co-predicate we use in this example is the built-in equality on co-datatype values. Intuitively, we can understand the proof as obtaining the co-inductive hypothesis of the proof goal (that is, with *rest* instead of xs) and adding some (automatically constructed) proof glue to reach the proof goal. Note the striking similarity between the inductive situation in Fig. 4 and the co-inductive situation in Fig. 9.

3.2 Prefix Methods

To understand why the code in Fig. 9 is a sound proof, let us now describe the details of the desugaring of co-methods. In analogy to how a **copredicate** declaration defines both a co-predicate and a prefix predicate, a **comethod** declaration defines both a co-method and *prefix method*. In the call graph, the cluster containing a co-method must contain only co-methods and prefix methods, no other

```
function SAppend(xs: Stream, ys: Stream): Stream
{
  match xs
  case SNil ⇒ ys
  case SCons(x, rest) ⇒ SCons(x, SAppend(rest, ys))
}

comethod SAppendIsAssociative(xs: Stream, ys:Stream,
                              zs: Stream)
  ensures SAppend(SAppend(xs, ys), zs) =
          SAppend(xs, SAppend(ys, zs));
{
  match xs {
    case SNil ⇒
    case SCons(x, rest) ⇒
      SAppendIsAssociative(rest, ys, zs);
  }
}
```

Figure 9. The definition of append on streams and the co-method that states and proves the associativity of append are almost exactly the same as the ones for finite lists, cf. Fig. 4. The only difference is the lack of **decreases** clauses and the use of a **comethod**. Recall that equality on co-datatypes, written =, is a built-in co-predicate. 8pll2 »

methods. By decree, a co-method and its corresponding prefix method are always placed in the same cluster. Both co-methods and prefix methods are always ghosts.

The prefix method is constructed from the co-method by

- adding a parameter _k of type **nat** to denote the prefix length,
- replacing in the co-method's postcondition the positive co-friendly occurrences of co-predicates by corresponding prefix predicates, passing in _k as the prefix-length argument,
- prepending _k to the (typically implicit) **decreases** clause of the co-method,
- replacing in the body of the co-method every intra-cluster call M(*args*) to a co-method by a call M#[_k − 1](*args*) to the corresponding prefix method, and then
- making the body's execution conditional on _k ≠ 0.

Note that this rewriting removes all co-recursive calls of co-methods, replacing them with recursive calls to prefix methods. These recursive call are, as usual, checked to be terminating. We allow the pre-declared identifier _k to appear in the original body of the co-method.[4]

We can now think of the body of the co-method as being replaced by a **forall** call, for every k, to the prefix method. By construction, this new body will establish the co-method's declared postcondition (on account of the $\mathcal{D}$ axiom, which we prove sound in Sect. 5, and remembering that only the positive co-friendly occurrences of co-predicates in the co-method's postcondition are rewritten), so there is no reason for the program verifier to check it.

```
// Desugaring automatically generated by Dafny
ghost method SAppendIsAssociative#[_k: nat]
     (xs: Stream, ys:Stream, zs: Stream)
  ensures SAppend(SAppend(xs, ys), zs) =#[_k]
          SAppend(xs, SAppend(ys, zs));
  decreases _k;
{
  if _k ≠ 0 {
    match xs {
      case SNil ⇒
      case SCons(x, rest) ⇒
        SAppendIsAssociative#[_k-1](rest, ys, zs);
    }
  }
}

ghost method SAppendIsAssociative(xs: Stream, ys:Stream,
                                  zs: Stream)
  ensures SAppend(SAppend(xs, ys), zs) =
          SAppend(xs, SAppend(ys, zs));
{
  forall k | 0 ≤ k {
    SAppendIsAssociative#[k](xs, ys, zs);
  }
}
```

Figure 10. Desugaring of the co-method SAppendIsAssociative in Fig. 9, showing the generated prefix method as well as the replacement of the body of the co-method. The first method uses induction on _k to establish its postcondition. Recall that $=^{\#}[\,\cdot\,]$ is the syntax for the prefix predicate of the built-in co-datatype equality. The postcondition of the second method follows from the **forall** statement and $\mathcal{D}$ axiom. The syntax for calling a prefix method sets apart the argument that specifies the prefix length; for this figure, we took the liberty of making up a coordinating syntax for the signature of the automatically generated prefix method. XxQK »

Figure 10 illustrates the result of desugaring the co-method SAppendIsAssociative from Fig. 9. In the recursive call of the prefix method, there is a proof obligation that the prefix-length argument _k − 1 is a natural number. Conveniently, this follows from the fact that the body has been wrapped in an **if** _k ≠ 0 statement. This also means that the postcondition must hold trivially when _k = 0, or else a postcondition violation will be reported. This is an appropriate design for our desugaring, because co-methods are expected to be used to establish co-predicates, whose corresponding prefix predicates hold trivially when _k = 0. (To prove other predicates, use an ordinary ghost method, not a co-method.)

It is interesting to compare the intuitive understanding of the co-inductive proof in Fig. 9 with the inductive proof in Fig. 10. Whereas the inductive proof is performing proofs for deeper and deeper equalities, the co-method can be understood as producing the infinite proof on demand.

3.3 Automation

Because co-methods are desugared into ghost methods whose postconditions benefit from induction, Dafny's usual induction tactic kicks in [21]. Effectively, it adds a **forall** statement at the beginning of the prefix method's body, invoking the prefix method recursively on all smaller tuples of arguments. Typically, the useful argument tuples are those with a smaller value of the implicit parameter _k and any other values for the other parameters, but the **forall** statement will also cover tuples with the same _k and smaller values of the explicit parameters.

Thanks to the induction tactic, the inductive ghost methods for proving associativity in Figs. 4 and 10 are verified automatically even if they are given empty bodies. So, co-method SAppendIsAssociative in Fig. 9 is also verified automatically

[4] Note, two places where co-predicates and co-methods are not analogous are: co-predicates must not make recursive calls to their prefix predicates, and co-predicates cannot mention _k.

```
comethod FivesUpPos(n: int)
  requires n > 0;
  ensures Pos(FivesUp(n));
  decreases 4 - (n - 1) % 5;
{
  if n % 5 = 0 { FivesUpPos#[_k-1](n + 1); }
  else { FivesUpPos#[_k](n + 1); }
}
```

Figure 11. A proof that, for any positive n, all values in the stream `FivesUp(`n`)` are positive. The proof uses both induction and co-induction. To illustrate what is possible, we show both calls as explicitly targeting the prefix method. Alternatively, the first call could have been written as a call `FivesUpPos(`$n+1$`)` to the co-method, which would desugar to the same thing and would more strongly suggest the intuition of appealing to the co-inductive hypothesis. plBHB »

even if given an empty body—it is as if Dafny had a tactic for automatic co-induction as well.

4. More Examples

In this section, we give further illustrative examples.

FivesUp The function FivesUp defined in Fig. 5 calls itself both recursively and co-recursively. To prove that FivesUp(n) satisfies Pos for any positive n requires the use of induction and co-induction together (which may seem mind boggling). We give a simple proof in Fig. 11. Recall that the **decreases** clause of the prefix method implicitly starts with _k, so the termination check for each of the recursive calls passes: the first call decreases _k, whereas the second call decreases the expression given explicitly. We were delighted to see that the **decreases** clause (copied from the definition of FivesUp) is enough of a hint to Dafny; it needs to be supplied manually, but the body of the co-method can in fact be left empty.

Filter The central issue in the FivesUp example is also found in the more useful *filter* function. It has a straightforward definition in Dafny:

```
function Filter(s: IStream): IStream
  requires AlwaysAnother(s);
  decreases Next(s);
{ if P(s.head)
  then ICons(s.head, Filter(s.tail))
  else Filter(s.tail)
}
```

In the **else** branch, Filter calls itself recursively. The difficulty is proving that this recursion terminates. In fact, the recursive call would not terminate given an arbitrary stream; therefore, Filter has a precondition that elements satisfying P occur infinitely often. To show progress toward the subsequent element of output, function Next counts the number of steps in the input s until the next element satisfying P.

The full example, which also proves some theorems about Filter, is found at 2slrL ». The filter function has also been formalized (with more effort) in other proof assistants, for example by Bertot in Coq [2].

Zip In Fig. 12, we define streams that are always infinite, some zip-related functions, and some properties of these (cf. [12]). The proof of EvenZipLemma is fully automatic, whereas the others require a single recursive call to be made explicitly. The **forall** statement inserted automatically by Dafny's induction tactic is in principle strong enough to prove each of the three lemmas, but the incompleteness of reasoning with quantifiers in SMT solvers makes the explicit calls necessary.

```
codatatype IStream<T> = ICons(head: T, tail: IStream)

function zip(xs: IStream, ys: IStream): IStream {
  ICons(xs.head, ICons(ys.head, zip(xs.tail, ys.tail)))
}
function even(xs: IStream): IStream {
  ICons(xs.head, even(xs.tail.tail))
}
function odd(xs: IStream): IStream { even(xs.tail) }
function bzip(xs: IStream, ys: IStream, f: bool)
         : IStream
{ if f then ICons(xs.head, bzip(xs.tail, ys, ¬f))
  else ICons(ys.head, bzip(xs, ys.tail, ¬f))
}

comethod EvenOddLemma(xs: IStream)
  ensures zip(even(xs), odd(xs)) = xs;
    { EvenOddLemma(xs.tail.tail); }
comethod EvenZipLemma(xs: IStream, ys: IStream)
  ensures even(zip(xs, ys)) = xs;
    { /* Automatic. */ }
comethod BzipZipLemma(xs: IStream, ys: IStream)
  ensures zip(xs, ys) = bzip(xs, ys, true);
    { BzipZipLemma(xs.tail, ys.tail); }
```

Figure 12. Some standard examples of combining and dividing infinite streams (cf. [12]). ZXDe »

```
copredicate True(s: IStream) { True(s.tail) }
comethod TrueLemma(s: IStream)
  ensures True(s);
{ TrueLemma(s.tail); }
comethod FalseLemma(s: IStream)
  ensures false;
// ERROR: postcondition violation
{ FalseLemma(s.tail); }
comethod BadProof(s: IStream)
  ensures True(s);
// ERROR: recursive call does not terminate
{ BadProof#[_k](s.tail); }
```

Figure 13. Since co-predicates are defined as greatest fix-points, `True(`s`)` holds for any s, as is confirmed by the fact that co-method TrueLemma verifies. A sanity check of soundness, trying to verify co-method FalseLemma does indeed result in an error message. Co-method BadProof fails to verify because the recursion is not well-founded. KsZl »

False Using the IStream type from the previous example, Fig. 13 shows basic sanity checks for co-inductive soundness. The part of the verification that fails for FalseLemma is the case where _k $= 0$. The figure also shows a buggy attempt to prove True(s), incorrectly passing in _k as the prefix length. Since this leads to infinite recursion, Dafny responds with an error message.

Co-predicate examples Figure 14 illustrates the co-friendly restriction.

```
copredicate NotIn<T>(n: T, s: IStream)
{ s.head ≠ n ∧ NotIn(n, s.tail) }

// There is an element not in the stream
predicate NotAll(s: IStream)
{ ∃ n • NotIn(n, s) }

// There is a non-zero number not in the stream; ERROR
copredicate NotAll'(n: int, s: IStream<int>)
{ if n = 0 then ∃ k • k ≠ 0 ∧ NotAll'(k, s)
  else n ≠ s.head ∧ NotAll'(n, s.tail) }

function Skip(n: int, s: IStream): IStream
{ if n ≤ 0 then s else Skip(n - 1, s.tail) }

// n occurs infinitely often in s
copredicate InfOften<T>(n: T, s: IStream)
{ (∃ k • Skip(k, s).head = n) ∧ InfOften(n, s.tail) }

// Alternative formulation - self-call under ∀ is OK
copredicate InfOften'<T>(n: T, s: IStream)
{ (∃ k • Skip(k, s).head = n) ∧
  (∀ k • InfOften'(n, Skip(k, s))) }

// ERROR
copredicate InfOften''<T>(n: T, s: IStream)
{ ∃ k • Skip(k, s).head = n ∧
        InfOften''(n, Skip(k, s)) }

// OK, quantification is bounded
copredicate QuiteOften(n: int, s: IStream<int>)
{ ∃ k • 0 ≤ k < n ∧ Skip(k, s).head = n ∧
        QuiteOften(n, Skip(k, s).tail) }
```

Figure 14. Examples of co-predicates which pass and fail the co-friendly restriction. NotAll' and InfOften'' fail because they call themselves recursively under an unbounded existential quantifier. Calls under no quantifier (NotIn, InfOften), under a universal quantifier (InfOften'), or under a bounded existential (QuiteOften) are permitted. zZCW »

Least and Greatest Fix-points There is an asymmetry in the design of Dafny. The **datatype** and **codatatype** declarations define inductive and co-inductive datatypes, respectively. The language provides both **predicate** and **copredicate** declarations, which may suggest that a **predicate** states an inductive property defined as a least fix-point, just like a **copredicate** states a co-inductive property defined as a greatest fix-point. This is the case with Fixpoint versus CoFixpoint in Coq, but in Dafny a **predicate** a simply a **function** that returns a boolean value—its recursive calls are checked to terminate and the predicate thus yields a particular value without the need to say anything about fix-points. Therefore, the first predicate in Fig. 15 yields a verification error, complaining about termination.

```
// ERROR: recursive call doesn't terminate
predicate IsFinite_LeastFixpoint(s: Stream)
{ match s
  case SNil ⇒ true
  case SCons(x, tail) ⇒ IsFinite_LeastFixpoint(tail)
}

predicate IsFinite(s: Stream)
{ ¬IsInfinite(s) }
copredicate IsInfinite(s: Stream)
{ match s
  case SNil ⇒ false
  case SCons(x, tail) ⇒ IsInfinite(tail)
}

predicate IsFinite'(s: Stream)
{ ∃ n • 0 ≤ n ∧ Tail(s, n) = SNil }
function Tail(s: Stream, n: nat): Stream
{ if s = SNil ∨ n = 0 then s else Tail(s.tail, n-1) }
```

Figure 15. Predicates in Dafny are always deterministic and well-defined, and thus have only one fix-point. Therefore, the first predicate above is not how to characterize finite streams; in fact, Dafny does not accept this definition. Finite streams can be characterized, however, either by negating a largest fix-point or by using an existential. H52v »

Iterates In a paper that shows co-induction being encoded in the proof assistant Isabelle/HOL, Paulson [31] defines a function Iterates(f, M) that returns the stream

$$M, f(M), f^2(M), f^3(M), \ldots$$

In Dafny syntax, the function is defined as

```
function Iterates<A>(M: A): Stream<A>
{ SCons(M, Iterates(f(M))) }
```

Paulson defines a function Lmap:

```
function Lmap(s: Stream): Stream
{ match s
  case SNil ⇒ SNil
  case SCons(a, tail) ⇒ SCons(f(a), Lmap(tail))
}
```

and proves that any function h satisfying

```
h(M) = SCons(M, Lmap(h(M)))
```

is indeed the function Iterates. This proof and all other examples from Paulson's paper can be done in Dafny, see S7aB ».

Wide Trees Let us consider defining a type of trees that are possibly infinite in width (that is, with a possibly infinite number of children) but finite in height. We start with the declaration

```
datatype Tree = Node(children: Stream<Tree>)
```

By itself, this declaration will allow structures that are infinite in height (the situation in Agda is similar [0]). The part of a Tree that can be inducted over is finite, in fact of size just 1 (recall from Sect. 1.3 that an inductive datatype value ends at its leaves, which in this case is the co-datatype value accessed via the destructor children). To restrict the height, we declare a predicate IsFiniteHeight:

```
predicate IsFiniteHeight(t: Tree)
{ ∃ n • 0 ≤ n ∧ LowerThan(t.children, n) }
copredicate LowerThan(s: Stream<Tree>, n: nat)
{ match s
  case SNil ⇒ true
  case SCons(t, tail) ⇒
    1 ≤ n ∧
    LowerThan(t.children, n-1) ∧ LowerThan(tail, n)
}
```

The use of a predicate to characterize an interesting subset of a type is typical in Dafny (also in the imperative parts of the language; for example, class invariants are just ordinary predicates [20]). QRqp »

Infinite Paths A simple example from Hur *et al.* [13] defines a predicate stating that a relation R extends infinitely from a given element x. We define it in Dafny as follows:

```
predicate InfPath(x: T)
{ ∃ p • Follows(x, p) }
copredicate Follows(x: T, path: IStream<T>)
{ R(x, path.head) ∧ Follows(path.head, path.tail) }
```

Note how this definition satisfies the co-friendly restriction, whereas the following one would not:

```
copredicate InfPath'(x: T)
{ ∃ y • R(x, y) ∧ InfPath'(y) }
```

The example also defines an extent of length n:

```
predicate Path(n: nat, x: T)
{ n = 0 ∨ ∃ y • R(x, y) ∧ Path(n-1, y) }
```

and proves that infinite extent implies any finite extent:

```
ghost method Theorem(n: nat, x: T)
  requires InfPath(x);
  ensures Path(n, x);
{ if n ≠ 0 {
    var p :| Follows(x, p);
    Theorem(n-1, p.head);
  }
}
```

Here, we have used Dafny's assign-such-that statement **var** `z :| Q(z)`, which checks ∃ `z • Q(z)`, introduces a variable `z`, and assigns to it an arbitrary value satisfying `Q(z)`. uPwg »

5. Soundness

In this section, we formalize and prove the connection between co-predicates and prefix predicates. More precisely, we prove that $\forall\, k \bullet\; P^{\#k}(\overline{x})$ is the greatest fix-point solution of equation defining $P(\overline{x})$.

Consider a given cluster of co-predicate definitions, that is, a strongly connected component of co-predicates:

$$P_i(\overline{x_i}) = C_i \quad \text{for } i = 0 \ldots n \qquad (0)$$

The right-hand sides (C_i) can reference functions, co-predicates, and prefix predicates from lower clusters, as well as co-predicates (P_j) in the same cluster. According to our restrictions in Sect. 2.2, the cluster contains only co-predicates, no prefix predicates or other functions; so, any prefix predicate referenced in C_i is necessarily from a lower cluster.

A cluster can be syntactically reduced to a single co-predicate, *e.g.*:

$$\begin{array}{rcl} P(i, \overline{x_0}, \ldots, \overline{x_n}) & = & 0 \le i \le n \;\wedge \\ & & ((i = 0 \;\wedge\; C_0\sigma) \;\vee\; \ldots \;\vee \\ & & (i = n \;\wedge\; C_n\sigma)) \\ \text{where } \sigma & = & [P_i := (\lambda\, \overline{x_i} \bullet\; P(i, \overline{x_0}, \ldots, \overline{x_n}))\,]_{i=0}^{n} \end{array} \qquad (1)$$

In what follows, we assume $P(x) = C_x$ to be the definition of P, where x stands for the tuple of arguments and C_x for the body above. Let:

$$\begin{array}{rcl} F_+(A) & = & \{x \mid C_x[P := A]\} \\ F(A) & = & \{x \mid \neg C_x[P := \neg A]\} \end{array} \qquad (2)$$

We liberally mix the notation of a set and its characteristic predicate using $\neg$ to mean set complement. We defined the semantics of a co-predicate to be the greatest fix-point of F_+. We are instead going to work with F and later use the following trivial lemma:

LEMMA 0. *$gfp(F_+) = \neg lfp(F)$*

DEFINITION 0. *A function f is* Scott-continuous *iff it is monotonic (*i.e., $A \subseteq B \implies f(A) \subseteq f(B)$*), and for any V and v such that $v \in f(V)$ there exists a finite V_0 such that $V_0 \subseteq V$ and $v \in f(V_0)$.*[5]

LEMMA 1 (Kleene fix-point theorem). *If f is Scott-continuous, then $lfp(f) = \bigcup_i f^i(\emptyset)$.*

LEMMA 2. *If $\psi = [\psi]_1$, $\perp$ doesn't occur in ψ, and $\psi[P := \neg V]$ evaluates to false, then there exists finite set $K(\psi, V)$, such that $\psi[P := \neg K(\psi, V)]$ evaluates to false.*

Proof: By induction on the structure of ψ.

0. if ψ is $\psi_0 \vee \psi_1$ then both ψ_0 and ψ_1 have to be false so we apply the lemma on both and let $K(\psi, V) = K(\psi_0, V) \cup K(\psi_1, V)$
1. if ψ is $\psi_0 \wedge \psi_1$ then ψ_i evaluates to false for $i = 0$ or $i = 1$ and $K(\psi, V) = K(\psi_i, V)$
2. if ψ is $\exists\, x \bullet\; e_0 \le x \le e_1 \implies \psi_0$ and e_0 evaluates to v_0 and e_1 to v_1, then we take $K(\psi, V) = \bigcup_{v_0 \le v \le v_1} K(\psi_0[x := v], V)$ as all $\psi_0[x := v]$ also evaluate to false
3. if ψ is $\forall\, x \bullet\; \psi_0$, then we find a witness v for which $\psi_0[x := v]$ is false and let $K(\psi, V) = K(\psi_i[x := v], V)$
4. if ψ is $P(t)$ and t evaluates to v then $K(\psi, V) = \{v\}$
5. otherwise P does not occur in ψ and we take $K(\psi, V) = \emptyset$

□

LEMMA 3. *F is Scott-continuous.*

Proof: F is monotonic because A occurs only positively in $C_x[P := A]$, and thus also in $\neg C_x[P := \neg A]$. Let us pick v and V such that $v \in F(V)$. By Lemma 2, we get $v \in F(K([C_x[x := v]]_1, V))$. □

DEFINITION 1. *Let $P^{\#}$ be the prefix predicate corresponding to P. We will write the prefix-length argument k as a superscript, as in $P^{\#k}$. The prefix predicates are defined inductively as follows:*

$$P^{\#0}(x) \quad \equiv \quad \top \qquad (3)$$

$$P^{\#k+1}(x) \quad \equiv \quad C[P := P^{\#k}] \qquad (4)$$

LEMMA 4. $\neg(x \in F^i(\emptyset)) \iff P^{\#i}(x)$

Proof: By induction on i. □

THEOREM 0.

$$x \in gfp(F_+) \iff \forall\, i \bullet\; P^{\#i}(x)$$

Proof:

$$\begin{array}{ll} & x \in gfp(F_+) \\ = & \quad \{ \text{ By Lemma 0. } \} \\ & \neg(x \in lfp(F)) \\ = & \quad \{ \text{ By Lemmas 1 and 3. } \} \\ & \neg(x \in \bigcup_i F^i(\emptyset)) \\ = & \quad \{ \text{ sets. } \} \\ & \forall\, i \bullet\; \neg(x \in F^i(\emptyset)) \\ = & \quad \{ \text{ Lemma 4. } \} \\ & \forall\, i \bullet\; P^{\#i}(x) \qquad \square \end{array}$$

[5] This definition captures computational essence of continuity—one only needs to look at finite input to produce an element of output. An alternative equivalent definition is often used where for any directed set D we have $\bigcup f(D) = f(\bigcup D)$.

6. Related Work

Most previous attempts at verifying properties of programs using co-induction have been limited to program verification environments embedded in interactive proof assistants. Early work includes an Isabelle/HOL package for reasoning about fix-points and applying them to inductive and co-inductive definitions [31]. The package was building from first principles and apparently lacked much automation. Later, a variant of the circular co-induction proof rule [33] was used in the CoCasl object-oriented specification system [12]. In CoCasl, as in the CIRC [24] prover embedded in the Maude term rewriting system, the automation is quite good.

Co-induction has long history in the Coq interactive proof assistant [7, 11]. A virtue of the standard co-induction tactic in Coq is that the entire proof goal becomes available as the co-induction hypothesis. One must then discipline oneself to avoid using it except in productive instances, something that is not checked until the final `Qed` command.

The language and proof assistant Agda [6, 28], which uses dependent types based on intuitionistic type theory, has some support for co-induction. Co-recursive datatypes and calls are indicated in the program text using the operators ∞ and $\sharp$ (see, *e.g.*, [0]). In Agda, proof terms are authored manually; there is no tactic language and no SMT support to help with automation.

Another programming language with support for induction and co-induction is Charity [8]. The language does not seem to have received much use, however. And although its design goals included assured termination of programs, the language did not use any program verifier to assure properties beyond termination.

Moore has verified the correctness of the compiler for the small language Piton [26]. The correctness theorem considers a run of k steps of a Piton program and shows that m steps of the compiled version of the program behave like the original, where m is computed as a function of k and k is an arbitrary natural number. One might also be interested in proving the compiler correctness for infinite runs of the Piton program, which could perhaps be facilitated by defining the Piton semantics co-inductively (*cf.* [22]). If the semantics-defining co-predicates satisfied our co-friendly restriction, then our $\mathcal{D}$ axiom would reduce reasoning about infinite runs to reasoning about all finite prefixes of those runs.

Our method of handling co-induction can be applied in any prover that readily handles induction. This includes verifiers like VCC [9] and VeriFast [14], but also interactive proof assistants. As shown in Fig. 11, induction and co-induction can benefit from the same automation techniques, so we consider this line of inquiry promising.

7. Conclusions

We have presented a method for reasoning about co-inductive properties, which requires only minor extensions of a verifier that already supports induction. In Dafny, the induction itself is built on top of off-the-shelf state-of-the-art first-order SMT technology [10], which provides high automation. In our initial experience, the co-inductive definitions and proofs seem accessible to users without a large degree of clutter. Even so, we suspect that further automation is possible once techniques for mechanized co-induction reach a maturity more akin to what is provided for induction by tools today (*e.g.*, [3, 17, 18, 21, 27, 29, 34]). With possible applications in both verifiers and other proof assistants, our work of making co-induction available in an SMT-based verifier takes a step in the direction of reducing the human effort in reasoning about co-induction.

Acknowledgments

During the course of this work, we have benefited from discussions with many colleagues who understand co-induction far better than we. We are grateful to all and mention here a subset: Jasmin Blanchette, Manfred Broy, Adam Chlipala, Ernie Cohen, Patrick Cousot, Jean-Christophe Filliâtre, Bart Jacobs (Nijmegen), Daan Leijen, Ross Tate.

References

[0] T. Altenkirch and N. A. Danielsson. Termination checking in the presence of nested inductive and coinductive types. Short note supporting a talk given at PAR 2010, Workshop on Partiality and Recursion in Interactive Theorem Provers, 2010. Available from `http://www.cse.chalmers.se/~nad/publications/`.

[1] B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Artificial Intelligence*. Springer, 2007.

[2] Y. Bertot. Filters on coinductive streams, an application to Eratosthenes' sieve. In P. Urzyczyn, editor, *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005*, volume 3461 of *Lecture Notes in Computer Science*, pages 102–115. Springer, Apr. 2005.

[3] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

[4] R. Bird and P. Wadler. *Introduction to Functional Programming*. International Series in Computing Science. Prentice Hall, 1992.

[5] S. Böhme and T. Nipkow. Sledgehammer: Judgement Day. In J. Giesl and R. Hähnle, editors, *Automated Reasoning, 5th International Joint Conference, IJCAR 2010*, volume 6173 of *Lecture Notes in Computer Science*, pages 107–121. Springer, July 2010.

[6] A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda — a functional language with dependent types. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer, Aug. 2009.

[7] A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, To appear. http://adam.chlipala.net/cpdt/.

[8] R. Cockett. The CHARITY home page. `http://pll.cpsc.ucalgary.ca/charity1/www/home.html`, 1996.

[9] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.

[10] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[11] E. Giménez. An application of co-inductive types in Coq: Verification of the alternating bit protocol. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES'95*, volume 1158 of *Lecture Notes in Computer Science*, pages 135–152. Springer, 1996.

[12] D. Hausmann, T. Mossakowski, and L. Schröder. Iterative circular coinduction for CoCasl in Isabelle/HOL. In M. Cerioli, editor, *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005*, volume 3442 of *Lecture Notes in Computer Science*, pages 341–356. Springer, 2005.

[13] C.-K. Hur, G. Neis, D. Dreyer, and V. Vafeiadis. The power of parameterization in coinductive proof. In R. Giacobazzi and R. Cousot, editors, *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13*, pages 193–206. ACM, Jan. 2013.

[14] B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Aug. 2008.

[15] B. Jacobs and J. Rutten. An introduction to (co)algebra and (co)induction. In D. Sangiorgi and J. Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, number 52 in Cambridge Tracts in Theoretical Computer Science, pages 38–99. Cambridge University Press, Oct. 2011.

[16] B. Jacobs, J. Smans, and F. Piessens. VeriFast: Imperative programs as proofs. In *VS-Tools workshop at VSTTE 2010*, Aug. 2010.

[17] M. Johansson, L. Dixon, and A. Bundy. Case-analysis for Rippling and inductive proof. In M. Kaufmann and L. C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010*, volume 6172 of *Lecture Notes in Computer Science*, pages 291–306. Springer, July 2010.

[18] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[19] C. Le Goues, K. R. M. Leino, and M. Moskal. The Boogie Verification Debugger (tool paper). In G. Barthe, A. Pardo, and G. Schneider, editors, *Software Engineering and Formal Methods — 9th International Conference, SEFM 2011*, volume 7041 of *Lecture Notes in Computer Science*, pages 407–414. Springer, Nov. 2011.

[20] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, Apr. 2010.

[21] K. R. M. Leino. Automating induction with an SMT solver. In V. Kuncak and A. Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation — 13th International Conference, VMCAI 2012*, volume 7148 of *Lecture Notes in Computer Science*, pages 315–331. Springer, Jan. 2012.

[22] X. Leroy. Coinductive big-step operational semantics. In P. Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 54–68. Springer, Mar. 2006.

[23] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[24] D. Lucanu and G. Rosu. Circ: A circular coinductive prover. In T. Mossakowski, U. Montanari, and M. Haveraaen, editors, *CALCO*, volume 4624 of *Lecture Notes in Computer Science*, pages 372–378. Springer, 2007. ISBN 978-3-540-73857-2.

[25] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982. ISBN 0387102353.

[26] J. S. Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):461–492, 1989.

[27] T. Nipkow, L. Paulson, and M. Menzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[28] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Sept. 2007.

[29] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification, 8th International Conference, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer, 1996.

[30] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Theoretical Computer Science, 5th GI-Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981.

[31] L. C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *Journal of Logic and Computation*, 7, 1997.

[32] S. Peyton Jones. *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 2003.

[33] G. Rosu and D. Lucanu. Circular coinduction: A proof theoretical foundation. In A. Kurz, M. Lenisa, and A. Tarlecki, editors, *CALCO*, volume 5728 of *Lecture Notes in Computer Science*, pages 127–144. Springer, 2009. ISBN 978-3-642-03740-5.

[34] W. Sonnex, S. Drossopoulou, and S. Eisenbach. Zeno: An automated prover for properties of recursive data structures. In C. Flanagan and B. König, editors, *Tools and Algorithms for the Construction and Analysis of Systems — 18th International Conference, TACAS 2012*, volume 7214 of *Lecture Notes in Computer Science*, pages 407–421. Springer, Mar.–Apr. 2012.