POPL 2018 Tutorial Proposal

Programming and Reasoning with Infinite Data in Isabelle/HOL

Mathias Fleury Andreas Lochbihler Andrei Popescu

Abstract

Recently we have endowed the proof assistant Isabelle/HOL with a powerful framework for programming with, and reasoning about, infinite objects such as streams and infinite-depth trees. It implements the paradigm of *total/productive coprogramming*, and is based on a modular design of coinductive datatypes. The tutorial will present this framework through examples taken from the field of programming languages.

Presenter's Short Bios

Mathias Fleury[1] is a PhD student at MPI-INF, supervised by Jasmin Blanchette and Christoph Weidenbach. He works on the Isabelle formalization of logical systems and programming languages.

Andreas Lochbihler[2] is a senior researcher at ETH Zürich, working on formalized programming languages and cryptography. He developed Isabelle's widely used Coinductive library. He proved correct a compiler for a large subset of concurrent Java and built the CryptHOL framework for cryptographic security proofs. Both projects rely heavily on codatatypes and total corecursive programming.

Andrei Popescu[3] is a senior lecturer at the Middlesex University London, working on proof assistant foundations and infrastructure and on information flow security. He designed and developed Isabelle's codatatype and corecursion mechanisms jointly with Jasmin Blanchette[4] and Dmitriy Traytel,[5] and the confidentiality-verified conference management system CoCon.

Background

Termination is a crucial property for the good behavior of programs, as it ensures responsiveness. Moreover, knowledge of termination allows for significantly more powerful rules for reasoning about programs.

Yet, some processes, notably operating systems and data stream processors, need to run forever, while still being responsive. An important class of such programs has been identified under the heading of *productive coprogramming* [1, 10]: These are programs that progressively reveal parts of their (potentially infinite) output, in such a way that they never become unresponsive, i.e., always eventually something is being revealed. For example, given a type of infinite streams constructed by infinitely many applications of ':', as in $x_1 : x_2 : x_3 : \ldots$, the definition $\mathsf{natsFrom}\ n = n : \mathsf{natsFrom}\ (n + 1)$ falls within this fragment, since each call to natsFrom produces one constructor before entering the nested call. The question on when an infinite-data manipulating program is productive can be non-trivial. For example, in the lambda calculus with recursive types $\mu X.\, T$, the mapping tree from syntactic recursive types to a possibly infinite tree type is (partly) given by

$$\mathsf{tree}\ (\mu X.\, T) = [X \mapsto \mathsf{tree}\ (\mu X.\, T)]\, T$$

i.e., T with X replaced by $\mathsf{tree}\ (\mu X.\, T)$. This specification is productive if and only if all occurrences X in T sit within an arrow type.

[1] https://people.mpi-inf.mpg.de/~mfleury
[2] http://www.infsec.ethz.ch/people/andreloc.html
[3] http://andreipopescu.uk
[4] https://people.mpi-inf.mpg.de/~jblanche
[5] http://people.inf.ethz.ch/trayteld

Standard programming languages do not guarantee termination or productivity. In other words, they allow users to write "sloppy," potentially unresponsive programs. By contrast, in a termination- and productivity-aware environment, such as a proof assistant, we must carefully distinguish between the well-founded inductive (or algebraic) datatypes and the non-well-founded coinductive (or coalgebraic) datatypes—often simply called *datatypes* and *codatatypes*, respectively. *Recursive* functions consume datatype values, peeling off constructors as they proceed; *corecursive* functions produce codatatype values, consisting of finitely or infinitely many constructors. And in the same way that *induction* is available as a proof principle to reason about datatypes and terminating recursive functions, *coinduction* supports reasoning about codatatypes and productive corecursive functions.

Many useful programs combine recursion with corecursion, which makes the checking of their productivity even more challenging. For example, is the function trace productive? It computes the possibly infinite list of all states of a while program run, where ++ concatenates two possibly infinite lists and last returns the last element of a finite list.

$$\begin{aligned}
\mathsf{trace}\,(\mathtt{skip}, s) &= [s] \\
\mathsf{trace}\,(\mathtt{x} := \mathtt{e}, s) &= [s(x \mapsto \mathsf{eval}\,(e, s))] \\
\mathsf{trace}\,(c\ ;\ c', s) &= (\mathsf{let}\ S = \mathsf{trace}\,(c, s)\ \mathsf{in}\ S +\!\!+ \mathsf{trace}\,(c', \mathsf{last}\ S)) \\
\mathsf{trace}\,(\mathtt{if}\ (b)\ c\ \mathtt{else}\ c', s) &= (\mathsf{if}\ \mathsf{eval}\,(b, s)\ \mathsf{then}\ \mathsf{trace}\,(c, s)\ \mathsf{else}\ \mathsf{trace}\,(c', s)) \\
\mathsf{trace}\,(\mathtt{while}\ b\ \mathtt{do}\ c\ \mathtt{od}, s) &= \mathsf{trace}\,(\mathtt{if}\ (b)\ c\ ;\ \mathtt{while}\ b\ \mathtt{do}\ c\ \mathtt{od}\ \mathtt{else}\ \mathtt{skip}, s)
\end{aligned}$$

Objectives and Covered Topic

The challenge of productive coprogramming is to have a framework that gives users the freedom to *write a wide variety of programs*, while at the same time *automatically guaranteeing productivity*. In recent years, in joint work with Blanchette and Traytel, we have taken up this challenge in the proof assistant Isabelle/HOL. We have extended the assistant with a mechanism to specify codatatypes, write productive corecursive functions on them, and reason coinductively about these functions. Our approach offers wide flexibility and modularity, based on two novel ideas in proof assistant technology:

Rich types: (Co)datatypes are viewed not only as collections of elements, but as rich structures featuring map functions (mappers) and relation-lifting functions (relators) [4, 9], which are maintained automatically [3, 6]: When users define a new (co)datatype, the system also defines the additional structure and proves characteristic polytypic theorems, e.g., that mappers distribute over composition.

Friendly, self-improving corecursion: The expressiveness of guaranteed-productive corecursion, encoded in a corecursion combinator, is not fixed, but evolves through the interaction with the users—newly defined functions can be registered as corecursive friendly, which makes them acceptable as participants in corecursive calls [2, 5].

In this three-hour tutorial, we will introduce the ideas underlying our approach in a hands-on fashion, by means of examples and exercises programmed in Isabelle/HOL. We will proceed gradually, starting with warm-up examples that involve streams and lazy lists and continue with more advanced material on branching structures (such as infinite-depth trees) applied to the semantics of programming. Isabelle/HOL features an ML-like language, understandable by anyone familiar with functional programming. In addition to user-level programs, we will present the evolving background infrastructure (mappers, relators, corecursion combinators) maintained by the system and show how it significantly streamlines future definitions. We will also show some Isabelle proofs by coinduction, which help establish desirable properties of productive programs, such as program equivalences.

At the end of the tutorial, the attendees will master the basics of productive coprogramming and will be able to start writing their own programs for infinite data processing, with applications to formalizing programming language semantics (e.g., to accompany their next POPL papers).

Who Should Attend and Why

The tutorial will target researchers having some experience with a typed functional programming language. Knowledge of a proof assistant and a basic understanding of coinduction are helpful for appreciating some finer points, but are not required.

Traditionally, productive coprogramming has only been supported by proof assistants based on Dependent Type Theory (DTT), such as Agda and Coq. These systems are more familiar to the programming language community, where they have been extensively advertised through tutorials, including at POPL. So why try our framework as an alternative to what Agda and Coq have to offer?

1. For non-dependently typed programs, the expressive power and automation of our corecursion scheme beat those of Coq by a wide margin and compare well with Agda's. Our scheme allows the user to freely write productive programs, rather than having to encode them in a rigid format or clutter them with additional annotations and proofs.

2. On the type specification level, our approach allows the flexible nesting of datatypes in codatatypes and vice versa. Moreover, it accommodates not only syntactically predetermined "free" types as in DTT, but also permutative types such as sets or bags, and even probability distributions—all very useful in programming language semantics. This flexibility is achieved on a *plug and play* basis: Any polymorphic type that has the right structure (mappers, relators, etc.) and is suitably bounded obtains clearance to participate in any (co)datatype definition.

3. Nested (co)datatypes come with corresponding nested (co)recursion principles, which are offered for free by the framework—whereas to achieve these in Agda and Coq users need to produce quite a few ad hoc auxiliary definitions and proofs, displaying significant ingenuity (as explained, e.g., in the "Nested Inductive Types" section in [7]).

4. Our framework is extremely logically safe, as it reduces all corecursive schemes to the non-(co)recursive primitives of Higher Order Logic. This guarantees that no inconsistency is being introduced, *even if the involved tools have bugs*—avoiding situations where seemingly harmless axioms of the logic become inconsistent due to a buggy termination/productivity checker [8].

Previous Tutorials

Two other tutorials, both titled "Certified (Co)programming with Isabelle/HOL," have been scheduled in 2017. The first[6] took place at CADE 2017 in Gothenburg on August 7 for a full day, with 15 registered participants. The second[7] is upcoming at ICFP 2017 in Oxford on September 3, will take place for half a day and has 24 registered participants. The topic of the POPL tutorial is partly overlapping with that of these other two. However, at CADE the focus was on coinductive proofs and at ICFP on functional pearls displaying certified productivity—by contrast, at POPL we will introduce the ideas by examples concerning formalized semantics of programming languages.

References

[1] R. Atkey and C. McBride. Productive coprogramming with guarded recursion. In *ICFP '13*, pp. 197–208. ACM, 2013.

[2] J. C. Blanchette, A. Bouzy, A. Lochbihler, A. Popescu, and D. Traytel. Friends with benefits: Implementing corecursion in foundational proof assistants. In *ESOP 2017*, LNCS, pp. 111–140. Springer, 2017.

[3] J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel. Truly modular (co)datatypes for Isabelle/HOL. In *ITP 2014*, vol. 8558 of *LNCS*, pp. 93–110. Springer, 2014.

[4] J. C. Blanchette, F. Meier, A. Popescu, and D. Traytel. Foundational nonuniform (co)datatypes for higher-order logic. In *LICS 2017*, vol. 10201 of *LNCS*, pp. 1–12. Springer, 2017.

[5] J. C. Blanchette, A. Popescu, and D. Traytel. Foundational extensible corecursion. In *ICFP 2015*, pp. 192–204. ACM, 2015.

[6] J. C. Blanchette, A. Popescu, and D. Traytel. Witnessing (co)datatypes. In *ESOP 2015*, vol. 9032 of *LNCS*, pp. 359–382. Springer, 2015.

[7] A. Chlipala. Inductive types in Coq. `http://adam.chlipala.net/cpdt/html/InductiveTypes.html`.

[8] M. Dénès. [Coq-Club] Propositional extensionality is inconsistent in Coq, 2013. Archived at `https://sympa.inria.fr/sympa/arc/coq-club/2013-12/msg00119.html`.

[9] D. Traytel, A. Popescu, and J. C. Blanchette. Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In *LICS 2012*, pp. 596–605. IEEE Comput. Soc., 2012.

[10] D. A. Turner. Elementary strong functional programming. In *FPLE '95*, vol. 1022 of *LNCS*, pp. 1–13. Springer, 1995.

[6] `http://www.cade-26.info`

[7] `http://icfp17.sigplan.org/track/icfp-2017-tutorials#event-overview`