Proof Assistants as Smart Programming Languages

Andrei Popescu

[1] Department of Computer Science, Middlesex University London, UK
[2] Institute of Mathematics Simion Stoilow of the Romanian Academy, Bucharest, Romania

Abstract. Proof assistants are tools specialized in helping humans prove theorems. What is perhaps less known is that they are also useful as programming environments. In this short text, I give a flavor of how a proof assistant can help write more reliable programs (e.g., guaranteed to be terminating or productive) without requiring the user to write proofs. I also give pointers to the work of my collaborators and me on improving the programming experience in the proof assistant Isabelle/HOL

In a typical proof assistant, a user can write commands like:

$$\mathsf{datatype}\ \alpha\ \mathsf{list} = \mathsf{Nil} \mid \mathsf{Cons}\ \alpha\ (\alpha\ \mathsf{list})$$

$$\mathsf{length\ xs}\ =\ \mathsf{case\ xs\ of\ Nil} \Rightarrow 0 \mid \mathsf{Cons\ x\ ys} \Rightarrow 1 + \mathsf{length\ ys}$$

Such commands are familiar to functional programmers, looking like usual recursive datatype and function defined in a functional programming language such as ML or Haskell. However, a proof assistant has a different take at such commands.

1 Theorem-Based Programming

For example, higher-order logic (HOL), the underlying logic of many successful proof assistants,[3] supports natively neither datatypes nor recursion—but only plain, nonrecursive definitions.

So how are such recursive definitions bootstrapped? *By automatically performing several nonrecursive definitions and proving several theorems.* Thus, when the user writes a datatype command like the above, the system does the following in response:

- Starts with the functor $(\alpha, \beta)\ \mathsf{F} = \mathsf{unit} + \alpha \times \beta$
- Builds its least fixpoint (initial algebra) $\alpha\ \mathsf{list} \simeq (\alpha, \alpha\ \mathsf{list})\ \mathsf{F}$
- Splits the fixpoint bijection into constructors Nil and Cons
- Proves the familiar properties: constructor injectivity, constructor-based induction and recursion principles, etc.

[3] These include HOL4 [25], HOL Light [14], ProofPower/HOL [2], HOL Zero [1], as well as my favorite, Isabelle/HOL [18, 19]—which actually implements a slight extension of HOL [16, 17], enabling Haskell-style type classes [20].

All this happens automatically, in the background, without the user needing to know. In Isabelle/HOL, 22000 lines of ML implement this functionality, covering both inductive and coinductiove datatypes (the latter also known as codatatypes).

Moreover, in response to a recursive function specification like the above for length, the system reacts as follows:

- Analyzes the call graph and proves that it is well founded
- Defines length using a well-founded recursion combinator
- Proves the above recursive equation as a theorem

In short, everything is reduced to nonrecursive definitions in HOL. I call such a programming model *theorem-based programming*, because the reduction proceeds by proving theorems.

A benefit of theorem-based programming is the availability of a significant amount of structure and static information on the defined types and programs. In particular, users obtain for free many useful polytypic operations on datatypes, such as map functions and "forall" predicates, as well as termination or productivity knowledge about their programs. This knowledge base is highly flexible and extendable, especially if the user is willing to do not only programming, but also a bit of proving. However, it should be insisted that a proof assistant is already useful to "ordinary" programmers, as it provides many facts automatically.

2 Theorems Versus Axioms

But why should one prefer theorem-based to the lighter[4] *axiom*-based programming? For example, when the user writes a recursive equation such as that of list, why not simply accept it as a new axiom of the system, say, after some syntactic checks, e.g., that all recursive calls are guarded? Why go further and establish well-foundedness of the call graph and then produce a nonrecursive definition in the background? There are two reasons. First, because the axiomatic approach is risky: Getting the checks slightly wrong leads to inconsistencies in the logic. Second, and equally importantly, because the axiomatic approach is inflexible: When the syntactic checks fail, the users cannot do anything, even if they know their recursion is correct. Consider the following specification of quicksort:

$$\mathsf{qsort}\ (\mathsf{Cons}\ x\ xs)\ =\ \mathsf{qsort}\ (\mathsf{filter}\ (<x)\ xs)\ ++\ [x]\ ++\ \mathsf{qsort}\ (\mathsf{filter}\ (\geq x)\ xs)$$

Without knowing the semantics of filter, a typical syntactic check must reject this definition—for all we know, filter could increase the size of its input list. In theorem-based programming, the system can ask the user for a hint or employ an existing fact about filter from the knowledge base, and then accept the definition.

In summary, theorem-based programming is the safest way to achieve *reliable* and *flexible* executable specifications in a proof assistant. Existing proof assistant tools realize these desiderata to different degrees—as discussed in the excellent recent survey [12], covering some of the most important players in the field: Agda [11], Coq [3], and the HOL-based proof assistants.

[4] That is, lighter for the proof assistant designers and implementors, not for the end users.

3 Achieving Flexibility

The HOL-based proof assistants are famously reliable thanks to reducing everything to a minimalistic logic kernel. In Isabelle/HOL [18, 19], we have recently also achieved significant flexibility:

- The defined datatypes can be inductive or coinductive [26], free or permutative [13, 22–24], and can be freely mixed and nested [5, 7, 9, 10].
- The functions defined on these datatypes can flexibly recurse [15], corecurse [4, 8], and even mix recursion and corecursion [4, 8].

These features required state-of-the art category theory, as well as mechanisms for customizing the abstract results to concrete cases—our motto was "employ category theory in the background, but do not expose the end user to it." There are two concepts behind this flexibility (achieved without compromising safety):

- *rich datatypes*, stored not as mere "types," i.e., flat collections of elements, but as functors and relators with additional structure and theorems [26] and with controlled size [6]—this enables modular constructions, preserving natural abstraction barriers
- *intelligent (co)recursors*, learning from their interaction with the users—this enables the system to become increasingly permissive with its allowed (co)recursion patterns [4, 8] and associated (co)induction principles [4, 8, 21]

4 Conclusion

Proof assistants are smart programming languages, in that they analyze each new user-specified datatype and program and integrate them in a knowledge base, fueled by proving theorems. This knowledge base offers substantial services to the programmer: It guarantees that programs terminate or are productive, and automatically defines many useful functions for datatypes. A wealth of additional static knowledge is made available to programmers who are willing to reach out and prove some basic properties of their programs. I wholeheartedly invite programmers to try Isabelle/HOL, which is one of the smartest (and friendliest) proof assistants ever to walk the earth.

Acknowledgment I gratefully acknowledge support from the UK Engineering and Physical Sciences Research Council (EPSRC) starting grant "VOWS: Verification of Web-based Systems" (EP/N019547/1).

References

1. 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)
2. Arthan, R.D.: Some mathematical case studies in ProofPower–HOL. In: Slind, K. (ed.) TPHOLs 2004 (Emerging Trends). pp. 1–16 (2004)
3. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions. Springer (2004)

4. Blanchette, J.C., Bouzy, A., Lochbihler, A., Popescu, A., Traytel, D.: Friends with benefits - implementing corecursion in foundational proof assistants. In: ESOP 2017. pp. 111–140 (2017)
5. 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)
6. Blanchette, J.C., Popescu, A., Traytel, D.: Cardinals in Isabelle/HOL. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 111–127. Springer (2014)
7. Blanchette, J.C., Popescu, A., Traytel, D.: Unified classical logic completeness: A coinductive pearl. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS, vol. 8562, pp. 46–60. Springer (2014)
8. 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)
9. Blanchette, J.C., Popescu, A., Traytel, D.: Soundness and completeness proofs by coinductive methods. J. Autom. Reasoning 58(1), 149–179 (2017)
10. Blanchette, J.C., Traytel, D., Popescu, A.: Foundational nonuniform (co)datatypes for higher-order logic. In: LICS 2017. IEEE Computer Society (2017), to appear
11. 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)
12. Bove, A., Krauss, A., Sozeau, M.: Partiality and recursion in interactive theorem provers - an overview. Mathematical Structures in Computer Science 26(1), 38–88 (2016)
13. Gheri, L., Popescu, A.: A general formalized theory of syntax with bindings. In: ITP 2017 (2017), to appear
14. 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)
15. Krauss, A.: Partial and nested recursive function definitions in higher-order logic. J. Autom. Reasoning 44(4), 303–336 (2010)
16. Kunčar, O., Popescu, A.: A consistent foundation for Isabelle/HOL. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, Springer (2015)
17. Kuncar, O., Popescu, A.: Comprehending Isabelle/HOL's consistency. In: ESOP 2017. pp. 724–749 (2017)
18. Nipkow, T., Klein, G.: Concrete Semantics: With Isabelle/HOL. Springer (2014)
19. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer (2002)
20. Nipkow, T., Snelting, G.: Type classes and overloading resolution via order-sorted unification. In: Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings. pp. 1–14 (1991)
21. Popescu, A., Gunter, E.L.: Incremental pattern-based coinduction for process algebra and its Isabelle formalization. In: Ong, C.H.L. (ed.) FoSSaCS 2010. pp. 109–127 (2010)
22. Popescu, A., Gunter, E.L.: Recursion principles for syntax with bindings and substitution. In: ICFP 2011. pp. 346–358 (2011)
23. Popescu, A., Gunter, E.L., Osborn, C.J.: Strong normalization of System F by HOAS on top of FOAS. In: LICS 2010. pp. 31–40. IEEE Computer Society (2010)
24. Schropp, A., Popescu, A.: Nonfree datatypes in Isabelle/HOL - animating a many-sorted metatheory. In: CPP. pp. 114–130 (2013)
25. 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)
26. 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)