Teaching Functional Programmers Logic and Metatheory

Frederik Krogsdal Jacobsen Jørgen Villadsen
DTU Compute - Department of Applied Mathematics and Computer Science - Technical University of Denmark

We present a novel approach for teaching logic and the metatheory of logic to students who have some experience with functional programming. We define concepts in logic as a series of functional programs in the language of the proof assistant Isabelle/HOL. This allows us to make notions which are often unclear in textbooks precise, to experiment with definitions by executing them, and to prove metatheoretical theorems in full detail. Our experience is that students grasp the meaning of programs quickly, and appreciate the precision available in the mechanized definitions and proofs.

1 Introduction

Logic is the foundation on which all of mathematics and computer science rests, and many undergraduate computer science programs therefore include an introductory course on logic. The logical systems introduced in such a course can be applied to databases, domain-specific languages, artificial intelligence, computer security, formal verification, and many other topics. At the Technical University of Denmark (DTU) we teach logic alongside logic programming in Prolog in a late-stage undergraduate course in addition to discrete mathematics taught in the first semester of the study program. Undergraduate courses like ours give students a basic understanding of logic and just enough knowledge to start applying basic logical systems in their work. But for students who really need to work with logic and design their own systems, this is not enough: they also need a good understanding of the metatheory of logic, i.e. why the systems work and how to prove that they do. Proofs about logical systems are fraught with possibilities for subtle mistakes of understanding, and it is our experience that many students never really "get" how the logical systems, and the proofs about them, work. Additionally, many textbooks define logical systems in terms of informal set theory and omit parts of their proofs (sometimes sweeping major complexities such as binders and substitution under the rug), which leads to difficulties in actually applying the theory when implementing real systems. For many students, implementing logical systems in their projects (and making sure that they are correct) can thus seem like an insurmountable challenge.

We have recently begun giving a graduate course on automated reasoning with course material implemented in the proof assistant Isabelle/HOL. This year the course started with 80 students. Isabelle/HOL is the higher-order logic version of the generic proof assistant Isabelle. Briefly, we can consider higher-order logic as the sum of functional programming and logic. Isabelle/HOL allows us to write definitions as functional programs and to formally prove properties of these programs. The proof assistant continuously checks that our proofs are correct, thus making it impossible to neglect any complexities. Additionally, Isabelle/HOL allows us to automatically export appropriate definitions into "real" functional languages such as Haskell, Scala and Standard ML, whereby they can be integrated into larger systems.

By defining the logical systems we teach within Isabelle/HOL, we are thus able to give precise and executable definitions of every aspect of the systems, and our proofs of metatheoretical properties such as soundness and completeness of systems relate directly to these precise definitions and are verified by the proof assistant. Our experience is that students appreciate this: if they are in doubt about what something means, they can look up a precise definition and even execute it on examples of their own choosing to gain further understanding, and this also makes it clear how to implement the concepts in practice.

We contribute:

- functional implementations of several logical systems which can be used to teach topics such as sequent calculus, natural deduction, de Bruijn indices, and algorithms for automatic theorem proving.
- formally verified proofs of common properties such as soundness and completeness for the systems mentioned above.

2 Related Work

There are of course numerous textbooks on logic, and several which include implementations of many of their definitions. Examples include books by Harrison [8], Ben-Ari [2], and Doets and van Eijck [4], which all contain implementations in various programming languages. Unfortunately, these implementations are not connected directly with the definitions and proofs in the books, and it is thus sometimes unclear how the programs really relate to the definitions in the text. Additionally, the proofs in these books are not verified by a proof assistant, and are about the informal definitions, not the programs themselves. Language, Logic and Proof [1] is a textbook and accompanying software package containing a number of small proof assistants designed to aid in teaching logic. While this means that students can rest assured that their exercise proofs are formally verified, the proofs in the book itself are not, and it is again sometimes unclear exactly how the software packages relate to the notions introduced in the text.

Some textbooks on logic do have formally verified proofs of their theorems, but these are typically not introductions to logic, but instead introductions to proof assistants that just happen to introduce some particular logic they are working in. One example is Coq'Art [3], which is an introduction to the Coq proof assistant and contains many formally verified proofs about logic and other topics. Unfortunately, such books, being written for a much more advanced audience, are not by themselves good introductions to logic, since they presuppose knowledge beyond the usual computer science program.

The approach of formalizing definitions as functional programs has successfully been used for a number of topics in computer science. The Software Foundations series [11] covers some basic logic, programming languages, formal verification, functional algorithms and separation logic with definitions and proofs in the Coq proof assistant. Concrete Semantics [10] is a textbook on programming language semantics with definitions and proofs in Isabelle/HOL. Functional Algorithms, Verified! [9] is an introduction to functional data structures and algorithms with definitions and proofs in Isabelle/HOL. Verified Functional Programming in Agda [13] is an introduction to functional programming itself, as well as functional algorithms, using the Agda programming language, which supports proofs about the programs written in it.

We have previously written about the contents of our courses [17], about teaching with Isabelle/Pure and Isabelle/HOL [5,6,16], and about individual formalizations [7,12,14,15], but not about the specifics and benefits of teaching logic and metatheory using functional programming to aid understanding.

3 A Glimpse of the Teaching Approach

As mentioned, our approach is to define the logical systems which are our objects of study as functional programs in Isabelle/HOL. Having done this, we can then define e.g. automatic theorem provers and other derived programs. Isabelle/HOL then allows us to prove results about the implementations directly, e.g. that our automatic theorem provers are sound and complete.

datatype *'a form*
  $= Pro\ 'a\ (\langle\cdot\rangle) \mid Falsity\ (\langle\bot\rangle) \mid Imp\ \langle 'a\ form\rangle\ \langle 'a\ form\rangle\ (\textbf{infixr}\ \langle\rightarrow\rangle\ 0)$

primrec *semantics* **where**
$\langle semantics\ i\ (\cdot n) = i\ n\rangle \mid$
$\langle semantics\ \text{-}\ \bot = False\rangle \mid$
$\langle semantics\ i\ (p \rightarrow q) = (semantics\ i\ p \longrightarrow semantics\ i\ q)\rangle$

abbreviation $\langle sc\ X\ Y\ i \equiv (\forall p \in set\ X.\ semantics\ i\ p) \longrightarrow (\exists q \in set\ Y.\ semantics\ i\ q)\rangle$

primrec *member* **where**
$\langle member\ \text{-}\ [] = False\rangle \mid$
$\langle member\ m\ (n\ \#\ A) = (m = n \vee member\ m\ A)\rangle$

lemma *member-iff* [*iff*]: $\langle member\ m\ A \longleftrightarrow m \in set\ A\rangle$
by (*induct A*) *simp-all*

primrec *common* **where**
$\langle common\ \text{-}\ [] = False\rangle \mid$
$\langle common\ A\ (m\ \#\ B) = (member\ m\ A \vee common\ A\ B)\rangle$

lemma *common-iff* [*iff*]: $\langle common\ A\ B \longleftrightarrow set\ A \cap set\ B \neq \{\}\rangle$
by (*induct B*) *simp-all*

function *mp* **where**
$\langle mp\ A\ B\ (\cdot n\ \#\ C)\ [] = mp\ (n\ \#\ A)\ B\ C\ []\rangle \mid$
$\langle mp\ A\ B\ C\ (\cdot n\ \#\ D) = mp\ A\ (n\ \#\ B)\ C\ D\rangle \mid$
$\langle mp\ \text{-}\ \text{-}\ (\bot\ \#\ \text{-})\ [] = True\rangle \mid$
$\langle mp\ A\ B\ C\ (\bot\ \#\ D) = mp\ A\ B\ C\ D\rangle \mid$
$\langle mp\ A\ B\ ((p \rightarrow q)\ \#\ C)\ [] = (mp\ A\ B\ C\ [p] \wedge mp\ A\ B\ (q\ \#\ C)\ [])\rangle \mid$
$\langle mp\ A\ B\ C\ ((p \rightarrow q)\ \#\ D) = mp\ A\ B\ (p\ \#\ C)\ (q\ \#\ D)\rangle \mid$
$\langle mp\ A\ B\ []\ [] = common\ A\ B\rangle$
by *pat-completeness simp-all*

termination
by (*relation* $\langle measure\ (\lambda(\text{-}, \text{-}, C, D).\ \sum p \leftarrow C\ @\ D.\ size\ p)\rangle$) *simp-all*

theorem *main*: $\langle (\forall i.\ sc\ (map \cdot A\ @\ C)\ (map \cdot B\ @\ D)\ i) \longleftrightarrow mp\ A\ B\ C\ D\rangle$
by (*induct rule*: *mp.induct*) (*simp-all*, *blast*, *meson*, *fast*)

definition $\langle prover\ p \equiv mp\ []\ []\ []\ [p]\rangle$

corollary $\langle prover\ p \longleftrightarrow (\forall i.\ semantics\ i\ p)\rangle$
unfolding *prover-def* **by** (*simp flip*: *main*)

Figure 1: An example Isabelle/HOL development defining a propositional logic and an automatic theorem prover for it. We begin by defining formulas with propositions, falsity, and implication as a datatype, then define semantics of formulas as a function. We then define an automatic theorem prover (function *mp*) for the system and prove that it terminates and is sound and complete. Note how every definition is a simple functional program and that Isabelle/HOL allows us to prove properties of these programs directly. Our students study this example very early on in our graduate course.

```
{-# LANGUAGE EmptyDataDecls, RankNTypes, ScopedTypeVariables #-}

module Scratch(Form, prover) where {

import Prelude ((==), (/=), (<), (<=), (>=), (>), (+), (-), (*), (/), (**),
  (>>=), (>>), (=<<), (&&), (||), (^), (^^), (.), ($), ($!), (++), (!!), Eq,
  error, id, return, not, fst, snd, map, filter, concat, concatMap, reverse,
  zip, null, takeWhile, dropWhile, all, any, Integer, negate, abs, divMod,
  String, Bool(True, False), Maybe(Nothing, Just));
import qualified Prelude;

data Form a = Pro a | Falsity | Imp (Form a) (Form a);

member :: forall a. (Eq a) => a -> [a] -> Bool;
member uu [] = False;
member m (n : a) = m == n || member m a;

common :: forall a. (Eq a) => [a] -> [a] -> Bool;
common uu [] = False;
common a (m : b) = member m a || common a b;

mp :: forall a. (Eq a) => [a] -> [a] -> [Form a] -> [Form a] -> Bool;
mp a b (Pro n : c) [] = mp (n : a) b c [];
mp a b c (Pro n : d) = mp a (n : b) c d;
mp uu uv (Falsity : uw) [] = True;
mp a b c (Falsity : d) = mp a b c d;
mp a b (Imp p q : c) [] = mp a b c [p] && mp a b (q : c) [];
mp a b c (Imp p q : d) = mp a b (p : c) (q : d);
mp a b [] [] = common a b;

prover :: forall a. (Eq a) => Form a -> Bool;
prover p = mp [] [] [] [p];

}
```

Figure 2: Haskell code generated by Isabelle/HOL from the example in Figure 1. Note that it is essentially a direct translation of the relevant Isabelle/HOL definitions, and that the proofs are not included. While the code is not pretty, a module such as this one can easily be integrated with other (handwritten) code to create a full application. This provides an example of how to implement and use the logical systems we cover in our course in practice.

3.1 Logical Systems and Provers as Functional Programs

As a simple example, Figure 1 contains a definition of classical propositional logic. We define a datatype of formulas (this is the so-called deep embedding approach), then introduce semantics as a function from truth value assignments (interpretations) and formulas to truth values (we also define a semantics function for sequents, *sc*). We can then define an automatic theorem prover (function *mp* for micro prover) for the system which works by breaking formulas up using a system similar to a sequent calculus. All of these definitions are simple functional programs, and students can thus understand them quite quickly when the ideas behind them are explained.

The programs can be executed directly inside of Isabelle/HOL, but can also be exported to standard functional programming languages, which allows us to use the concepts in practice. The Haskell program in Figure 2 has been automatically generated by Isabelle/HOL from the definitions in Figure 1. Note that only the functions themselves, and not the proofs about them, are present in the Haskell code.

3.2 Metatheory as Properties of Programs

Once we have defined a logical system in Isabelle/HOL, we can begin proving results about it. In Figure 1, we first prove that the functions *member* and *common* are correct by simple structural induction. Note that the proofs very much resemble the classic "The proof is by induction" often found in textbooks, but that the proof has been verified by Isabelle/HOL. This allows the reader to rest easy knowing that there is no hidden complexity in the proof. We then prove termination of our prover by noting that the sum of the sizes of the two last arguments decrease. Isabelle/HOL requires that all functions are total, and in this case the proof is complicated enough that we have to prove it manually, but simple enough that we can do so easily. Next, we prove that our automatic theorem prover returns true if and only if the provided sequent is valid (theorem *main*). This proof is by induction, and the automation in Isabelle/HOL is enough to handle all cases. Finally, we conclude that our prover returns true for a formula exactly when it is valid (true in all interpretations), which means that the prover is sound and complete.

4 Conclusion

We have presented our approach for teaching logic and metatheory using functional programming and demonstrated how logical concepts can be defined as simple functional programs about which we can prove metatheoretical results in a natural way. In our experience, this approach can help students understand the concepts more easily, and we have explained how our approach makes it easier for students to actually apply the concepts in real implementations.

For now, our course material consists of a series of lecture notes and corresponding formalizations and student exercise templates in Isabelle/HOL, and we combine this with excerpts from textbooks and tutorials. It would be great to have a comprehensive and coherent textbook written with this style of teaching in mind, but our course material has not yet stabilized enough for us to consider writing one. We would like to make clear that a book containing nothing but programs is not what we advocate; instead, we would like a textbook containing explanations based on, and corresponding to, the precise definitions in the functional programs, such that readers can always clarify any doubts about the "intuitive" explanations given in the text by referring to the programs implementing the definition. Similarly, we do not believe that textbooks should contain meticulous proofs going into every detail of every case, but that such proofs should be available to the reader if they are not convinced by the abbreviated arguments given in the text.

Acknowledgements

We thank Asta Halkjær From and Simon Tobias Lund for comments on drafts.

References

[1] David Barker-Plummer, Jon Barwise & John Etchemendy (2011): *Language, Proof and Logic*, 2. edition. Center for the Study of Language and Information.

[2] Mordechai Ben-Ari (2012): *Mathematical Logic for Computer Science*. Springer, doi:10.1007/978-1-4471-4129-7.

[3] Yves Bertot & Pierre Castéran (2004): *Interactive Theorem Proving and Program Development*. Springer, Berlin, Heidelberg, doi:10.1007/978-3-662-07964-5.

[4] Kees Doets & Jan van Eick (2012): *The Haskell Road to Logic, Maths and Programming*, 2. edition. College Publications.

[5] Asta Halkjær From, Alexander Birch Jensen, Anders Schlichtkrull & Jørgen Villadsen (2019): *Teaching a Formalized Logical Calculus*. In Pedro Quaresma, Walther Neuper & João Marcos, editors: *Proceedings 8th International Workshop on Theorem Proving Components for Educational Software, ThEdu@CADE 2019, Natal, Brazil, 25th August 2019*, *EPTCS* 313, pp. 73–92, doi:10.4204/EPTCS.313.5.

[6] Asta Halkjær From, Jørgen Villadsen & Patrick Blackburn (2020): *Isabelle/HOL as a Meta-Language for Teaching Logic*. In Pedro Quaresma, Walther Neuper & João Marcos, editors: *Proceedings 9th International Workshop on Theorem Proving Components for Educational Software, ThEdu@IJCAR 2020, Paris, France, 29th June 2020*, *EPTCS* 328, pp. 18–34, doi:10.4204/EPTCS.328.2.

[7] Asta Halkjær From, Anders Schlichtkrull & Jørgen Villadsen (2021): *A Sequent Calculus for First-Order Logic Formalized in Isabelle/HOL*. In Stefania Monica & Federico Bergenti, editors: *Proceedings of the 36th Italian Conference on Computational Logic - CILC 2021, Parma, Italy, September 7-9, 2021*, *CEUR Workshop Proceedings* 3002, CEUR-WS.org, pp. 107–121. Available at `http://ceur-ws.org/Vol-3002/paper7.pdf`.

[8] John Harrison (2009): *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, doi:10.1017/CBO9780511576430.

[9] Tobias Nipkow, Jasmin Blanchette, Manuel Eberl, Alejandro Gómez-Londoño, Peter Lammich, Christian Sternagel, Simon Wimmer & Bohua Zhan (2021): *Functional Algorithms, Verified!* Available at `https://functional-algorithms-verified.org/`.

[10] Tobias Nipkow & Gerwin Klein (2014): *Concrete Semantics*. Springer, Cham, doi:10.1007/978-3-319-10542-0.

[11] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg & Brent Yorgey (2017): *Software Foundations*. Electronic textbook. `http://www.cis.upenn.edu/~bcpierce/sf`.

[12] Anders Schlichtkrull, Jørgen Villadsen & Andreas Halkjær From (2018): *Students' Proof Assistant (SPA)*. In Pedro Quaresma & Walther Neuper, editors: *Proceedings 7th International Workshop on Theorem proving components for Educational software, ThEdu@FLoC 2018, Oxford, United Kingdom, 18 July 2018*, *EPTCS* 290, pp. 1–13, doi:10.4204/EPTCS.290.1.

[13] Aaron Stump (2016): *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan & Claypool, doi:10.1145/2841316.

[14] Jørgen Villadsen (2020): *Tautology Checkers in Isabelle and Haskell*. In Francesco Calimeri, Simona Perri & Ester Zumpano, editors: *Proceedings of the 35th Italian Conference on Computational Logic - CILC 2020, Rende, Italy, October 13-15, 2020*, *CEUR Workshop Proceedings* 2710, CEUR-WS.org, pp. 327–341. Available at `http://ceur-ws.org/Vol-2710/paper21.pdf`.

[15] Jørgen Villadsen, Andreas Halkjær From & Anders Schlichtkrull (2018): *Natural Deduction Assistant (NaDeA)*. In Pedro Quaresma & Walther Neuper, editors: *Proceedings 7th International Workshop on Theorem proving components for Educational software, ThEdu@FLoC 2018, Oxford, United Kingdom, 18 July 2018*, *EPTCS* 290, pp. 14–29, doi:10.4204/EPTCS.290.2.

[16] Jørgen Villadsen, Asta Halkjær From & Patrick Blackburn (2022): *Teaching Intuitionistic and Classical Propositional Logic Using Isabelle*. In João Marcos, Walther Neuper & Pedro Quaresma, editors: Proceedings 10th International Workshop on *Theorem Proving Components for Educational Software*, (Remote) Carnegie Mellon University, Pittsburgh, PA, United States, 11 July 2021, *Electronic Proceedings in Theoretical Computer Science* 354, Open Publishing Association, pp. 71–85, doi:10.4204/EPTCS.354.6.

[17] Jørgen Villadsen & Frederik Krogsdal Jacobsen (2021): *Using Isabelle in Two Courses on Logic and Automated Reasoning*. In João F. Ferreira, Alexandra Mendes & Claudio Menghi, editors: *Formal Methods Teaching*, Springer International Publishing, Cham, pp. 117–132, doi:10.1007/978-3-030-91550-6_9.