VeriTaS: Verification of Type System Specifications

Mechanizing Domain Knowledge about Progress and Preservation Proofs

Sylvia Grewe
TU Darmstadt, Germany
grewe@cs.tu-darmstadt.de

Abstract

Developing a type system with a soundness proof is hard. The VeriTaS project aims at simplifying the development of sound type systems through automation of soundness proofs and through automated derivation of efficient type checkers from sound type system specifications.

Within the VertiTaS project, I focus on developing an interface for the verification of progress and preservation proofs which shall automate standard parts of such proofs. To achieve this, I propose to identify recurring proof strategies in progress and preservation proofs from the literature, to develop a format for abstractly representing these proof strategies, and to mechanize them by connecting them to existing theorem provers.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; I.2.3 [*Artificial intelligence*]: Deduction and Theorem Proving

Keywords Type systems, type soundness, theorem proving

1. Motivation

Nowadays, researchers frequently develop type systems for programming languages, for example for domain-specific languages (DSLs) or for core calculi of general-purpose programming languages. Typically, most researchers in the area of programming languages agree that such type system developments should be accompanied by full soundness proofs. They also agree that one should use a theorem prover such as Isabelle [9], Coq [8], Twelf [1], etc. to derive the soundness proofs, since mistakes in pen-and-paper proofs are very common. Numerous solutions to the POPLMARK challenge [3] demonstrated that one can, in principle, use many different existing theorem provers for mechanizing the soundness proofs of type systems.

Despite these demonstrations, many researchers still resort to pen-and-paper formalizations and proofs of their type systems. Concrete concerns about using an existing verification tool which I gathered from these researchers include:

1. "It takes too long to develop a full proof using tool X."
2. "I do not want to print the formalization in the language of tool X in my paper. I want to present type system specifications in a notation that is common in the area of programming languages."
3. "I already have to develop an implementation in a general-purpose language of my type system anyway, because I want to be able to test, debug, and benchmark my type system - I do not want to develop a mechanized formalization in tool X in addition to that."

Here, concern 1 seems to be the main concern, while the other two seem to weigh slightly less heavy.

How can we make it more attractive for programming language researchers to mechanize soundness proofs of type systems? In particular, how can we reduce the effort of mechanizing these proofs?

2. Problem

One standard way from literature for formalizing type soundness is via progress and preservation theorems. Past research, notably the work by Wright and Felleisen [12], led to a unified structure of progress and preservation proofs, which one can study in detail in Pierce's TAPL [10]. This "unified structure" seems to be so well understood that some researchers do not even bother to completely spell it out in their publications. However, to the best of my knowledge, no one has attempted to formalize or mechanize this abstract domain knowledge about progress and preservation proofs.

I propose to mechanize this domain knowledge and to use it in combination with existing automated proof techniques, with the aim of reducing the effort of mechanizing standard progress and preservation proofs. Concretely, I focus on the following research questions:

(a) What are recurring strategies in progress and in preservation proofs? I will call such strategies *domain-specific* *proof strategies*, where the domain in my case are progress and preservation proofs.

(b) What is a useful format for *abstractly representing* domain-specific proof strategies? To be useful, the format should allow for representing a proof strategy such that it is applicable to several concrete progress and preservation proofs. Additionally, the format should be understandable by programming languages researchers who develop soundness proofs of type systems, i.e. by domain experts. If necessary, domain experts should be able to use the format to define their own domain-specific proof strategies.

(c) How can one *mechanize* a domain-specific proof strategy and its application in concrete progress and preservation proofs? How can the mechanized strategies interact with existing theorem proving techniques? And in particular, how can one mechanize these strategies so that programming language researchers are more willing to use the resulting implementation for mechanizing progress and preservation proofs?

Several existing verification tools provide formats for specifying abstract proof strategies. For example, Coq [8] provides the ltac language for defining custom proof tactics and proof search methods. Isabelle [9] allows for implementing customized proof tactics in ML within theory files. Additionally, there is an implementation of proof planning [11] within Isabelle, the IsaPlanner [4]. IsaPlanner includes implementations of various general-purpose plans such as rippling [2], which is a powerful technique for automating the proofs of certain induction steps. However, the usage of such languages and techniques typically requires rather deep knowledge about the internals of a theorem prover and about verification techniques. It is not obvious how one would encode a domain-specific proof strategy so that domain experts can easily understand the strategies and encode their own ones.

3. Approach

3.1 Identifying Domain-Specific Proof Strategies

To identify proof strategies that are domain-specific to progress and preservation proofs, I will study the respective proofs in the established literature and in publications such as TAPL [10]. My goal is to identify the most common techniques used, which may include induction schemes, templates for auxiliary lemmas, and common proof techniques for verifying individual induction cases.

Candidate techniques that I already identified are, for instance, structural induction on the syntax of the programming language in question, induction on typing derivations, case distinction on rules of the dynamic semantics, inversion lemmas, and lemmas which ”propagate” progress or preservation along to auxiliary functions used in the static and dynamic semantics of the language. An example for such a propagation lemma is the well-known substitution lemma from the soundness proof of the standard type system for the simply-typed lambda calculus (e.g. TAPL [10]), which ”propagates” the preservation property onto the substitution function.

This first part of my research project shall generate an informal list of domain-specific proof strategies, which can by itself already serve as a guide for other researchers who develop progress and preservation proofs.

3.2 Representing Domain-Specific Proof Strategies

In the second part of my research project, I will develop a format which allows for representing domain-specific proof strategies abstractly, that is, without concrete knowledge about the typing rules or the small-step rewrite rules of the dynamic semantics. The format shall, on the one hand, allow for representing domain-specific proof strategies so that they are applicable to different concrete type systems and dynamic semantics. On the other hand, the format shall remain comprehensible for domain experts, i.e. researchers who understand the structure of progress and preservation proofs.

To meet these criteria, I propose to represent domain-specific proof strategies as *open proof tree templates*: As described for example by Richardson and Bundy [11], one can represent a proof via a *proof tree*, whose root is the theorem to be proven and whose nodes represent subgoals arising from intermediate proof steps such as induction or case distinction or represent auxiliary lemmas. The edges from a parent node to its children represent a proof strategy such that a proof of the parent node follows from proofs of the child nodes. *Verified nodes* and *verified edges* represent subgoals or proof steps that have been verified by a theorem prover. A proof tree is *closed* if all its leaves are verified nodes and if all its edges are verified. A proof tree is *open* if it contains any unverified leaves or edges.

A proof tree *template* shall *abstract* over concrete domain-specific concepts in a proof tree. In the domain of progress and preservation proofs, such concepts are, for instance, typing rules or rewrite rules from the dynamic semantics. The abstraction of a domain-specific concept shall preserve any abstract information about the concrete concept which is necessary for an abstract representation of the domain-specific strategy. For example, for a typing rule, such information may include whether the rule is an introduction or an elimination rule, the number of premises, an abstract representation of the typing rule’s conclusion, etc. One of the main challenges of this part of my project will be to identify which abstract information about the domain-specific concepts involved in progress and preservation proofs is required for abstractly representing domain-specific proof strategies.

Representing domain-specific proof strategies as templates of open proof trees shall enable domain experts to understand the strategies and to develop their own domain-specific proof strategies if needed: The templates shall employ the terminology of progress and preservation proofs and abstract over concepts that are familiar to domain experts.

3.3 Mechanizing Domain-Specific Proof Strategies

I propose to mechanize domain-specific proof strategies within an infrastructure that can address all three concerns of programming language researchers from Section 1. Concretely, I propose to design an interface for the mechanization of soundness proofs of type systems in the form of an extensible library within the VeriTaS project [5]. The library shall be implemented within a general-purpose programming language: This way, programming language researchers are able to verify a type system specification by solely interacting with the library, in a programming language which they are likely to know already. Furthermore, library users will be able to extend the library based on their custom needs. For example, a user could add pretty-printing in LaTeX or add functionality for executing certain type system specifications.

VeriTaS shall comprise an internal domain-specific language for specifying syntax, dynamic semantics and type systems of programming languages and for proving properties on these specifications. It shall include an implementation of proof trees and domain-specific strategies as suggested in Section 3.2, together with an API for interacting with proof trees. Such an API includes for example functionality for triggering the automatic generation of closed or open proof trees, for inspecting proof trees, for manually calling specific proof strategies on inner nodes, for triggering the verification of leaves and edges, and for replacing nodes entirely with custom intermediate proof steps if necessary. VeriTaS shall interact with existing theorem provers such as Vampire [7] and Isabelle [9] for verifying edges and leaves of proof trees.

VeriTaS shall provide extension points for adding specification constructs, as well as additional functionality that addresses concerns 2 and 3 from Section 1. However, implementing the corresponding functionality shall not be the main focus of my research project.

An ideal candidate language for implementing the VeriTaS library is Scala: The Scala language is widely known and used in the programming languages community. Scala is well-suited for implementing expressive internal DSLs. Scala also provides numerous language features which will facilitate the design of extension points for the library, such as object orientation and generics. Finally, building the VeriTaS library in Scala will allow us to (re)use existing Scala code that connects to different verification tools. For example, the libisabelle[1] library allows for interacting with Isabelle via Scala, as successfully demonstrated by Hupel et al. [6].

4. Evaluation

I will focus on evaluating whether the domain-specific strategies decrease the user effort for proving progress and preservation, compared to using existing verification tools.

I plan to evaluate this by preparing different case studies consisting of type system specifications. I will formalize these specifications along with progress and preservation proofs once in VeriTaS, and once in Isabelle, using the Isar proof language. I will compare the number of individual user interactions against each other. That is, in VeriTaS, every manual modification of a proof tree will count as a user interaction, in Isabelle every proof command that modifies the subgoals, together with every lemma specification. Furthermore, provided I can find suitable candidates, I plan to conduct a small study where I will ask other programming language researchers or students to conduct progress and preservation proofs using VeriTaS and to rate their experience.

[1] `https://github.com/larsrh/libisabelle`

Acknowledgments

I thank my supervisors Mira Mezini and Sebastian Erdweg and my fellow PhD students Oliver Bracevac, Manuel Weiel, and Lars Hupel for helpful discussions. This work is supported by the European Research Council, grant No. 321217.

References

[1] The Twelf project. `http://twelf.org/`, 2014.

[2] Alan Bundy et al. Rippling: A heuristic for guiding inductive proofs. *Artif. Intell.*, 62(2):185–253, 1993.

[3] Brian E. Aydemir et al. Mechanized Metatheory for the Masses: The POPLMARK Challenge. In *Proceedings of International Conference on Theorem Proving in Higher Order Logics (TPHOL)*, pages 50–65. Springer-Verlag, 2005.

[4] L. Dixon and J. D. Fleuriot. Isaplanner: A prototype proof planner in isabelle. In *Proceedings of International Conference on Automated Deduction (CADE)*, pages 279–283, 2003.

[5] S. Grewe, S. Erdweg, P. Wittmann, and M. Mezini. Type systems for the masses: Deriving soundness proofs and efficient checkers. In *Proceedings of International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (ONWARD)*, pages 137–150. ACM, 2015.

[6] L. Hupel and V. Kuncak. Translating Scala Programs to Isabelle/HOL - System description. In *Proceedings of International Joint Conference on Automated Reasoning (IJCAR)*, 2016.

[7] L. Kovács and A. Voronkov. First-order theorem proving and Vampire. In *Proceedings of International Conference on Computer Aided Verification (CAV)*, pages 1–35. Springer, 2013.

[8] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL `http://coq.inria.fr`. Version 8.0.

[9] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer, 2002.

[10] B. C. Pierce. *Types and programming languages*. MIT press, 2002.

[11] J. Richardson and A. Bundy. Proof planning methods as schemas. *J. Symbolic Computation*, 11:1–000, 1999.

[12] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.