System Description: An Infrastructure for Combining Domain Knowledge with Automated Theorem Provers

Sylvia Grewe
Technische Universität Darmstadt
Germany

Sebastian Erdweg
Delft University of Technology
Netherlands

André Pacak
Technische Universität Darmstadt
Germany

Mira Mezini
Technische Universität Darmstadt
Germany

ABSTRACT

Computer science has seen much progress in the area of automated verification in the last decades. Yet, there are many domains where abstract strategies for verifying standard properties are well-understood by domain experts, but still not automated to a satisfactory degree. One example for such a domain are type soundness proofs. Being able to express domain-specific verification strategies using domain-specific terminology and concepts can help to narrow down this gap toward more automated verification.

We present the requirements, design, and implementation of a configurable verification infrastructure that allows for expressing domain knowledge about proofs and for interfacing with existing automated theorem provers and solvers to verify individual proof steps. As an application scenario for our infrastructure, we present the development of a standard type soundness proof for a typed subset of SQL.

CCS CONCEPTS

• **Theory of computation** → **Automated reasoning**; • **Software and its engineering** → *Software verification*; *Domain specific languages*;

ACM Reference Format:
Sylvia Grewe, Sebastian Erdweg, André Pacak, and Mira Mezini. 2018. System Description: An Infrastructure for Combining Domain Knowledge with Automated Theorem Provers. In *The 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18), September 3–5, 2018, Frankfurt am Main, Germany.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3236950.3236960

1 INTRODUCTION

Assume that you are an expert on how to verify a standard property in a specific domain, such as proving type soundness, proving liveness of a state machine, proving non-interference, solving a certain constraint-based problem etc. You know all about the general steps that are typically needed to solve your specific problem, you have an idea in which order to apply these steps, and you know the structure of the auxiliary lemmas that you need. Your goal is to automate proofs in your domain as much as possible. How could you approach this task today, and how far could you get?

Firstly, automated theorem provers (ATPs) such as Vampire [11] and SMT solvers (satisfiability modulo theories) such as Z3 [4] are able to automatically solve a large number of first-order proof problems from different domains. ATPs and SMT solvers typically solve proof problems that require generating models for quantified variables which have to satisfy given constraints. Typically, such problems require applying the given definitions from the problem specification in the correct order. This is a good start for trying to automate your verification task. But what if your problems are too complicated to be solved by ATPs or SMT solvers directly? This could happen either because your definitions are complex enough that the search space for proofs gets too large, or because solving your problems needs higher-order reasoning techniques such as induction, or because your problems require additional auxiliary lemmas, which ATPs and SMT solvers typically cannot synthesize by themselves.

Interactive theorem provers such as Isabelle [14] and Coq [6] enable the mechanization of large and complex proofs in higher-order logic, as well as the partial automation of such proofs by writing custom code for generating proof steps, so-called tactics. Projects that combine both of these worlds, such as Isabelle Sledgehammer [3] which lets Isabelle interface with different ATPs and SMT solvers, raise the degree of automation. So you could try to encode your domain-specific proof strategies via tactic languages such as ltac [5] or MTac [23] in Coq or Eisbach [13] in Isabelle.

Existing tactic languages typically operate as follows: You may match the current goal(s) on one or more goal schemas and then apply existing general-purpose tactics to obtain another goal or set of sub-goals. Additionally, you may use different combinators with the existing general-purpose tactics, such as conditional application of tactics (only applying a tactic if the current goal satisfies a certain condition), sequential or repeated application of tactics, etc. However, existing tactic languages typically do not enable you to do anything beyond manipulating the current goal state. Especially, you can neither synthesize auxiliary lemmas needed in your proof, nor access the AST (abstract syntax tree) of a problem specification, for example to inspect the definition of a particular function or predicate to determine the shape of an auxiliary lemma. Yet, both of these capabilities are crucial to approach fully automatic proof generation in many interesting domains. Another issue with existing tactic languages is that they require you to encode your

PPDP '18, September 3–5, 2018, Frankfurt am Main, Germany

ACM ISBN 978-1-4503-6441-6/18/09.
https://doi.org/10.1145/3236950.3236960

domain language and any domain-specific strategies via general-purpose concepts and strategies within Coq or Isabelle. That is, the domain-specific abstractions of your problem get obfuscated by prover-specific details.

To overcome the issues mentioned, we propose the design of a verification infrastructure which is configurable in the input format for problem specifications as well as in the format in which tactics and proof strategies can be specified. We propose to implement such a verification infrastructure within a general-purpose programming language so that users may design domain-specific languages (DSLs) or APIs for expressing their problem domains and build domain-specific strategies to automatically construct a proof for a domain-specific problem.

To achieve maximum flexibility when it comes to generating a proof, we propose to strictly separate two concerns that are often convoluted in existing systems: the construction of a *proof structure* and the verification of the individual *proof steps* in such a proof structure. Our infrastructure enables the generation of high-level proof structures which consist of various lower-level proof steps, such as performing an inductive step, equational reasoning, case distinctions, and lemma applications. The generated proof structure can be seen as a *recommendation* for a proof, i.e. it may not be fully complete, and may even contain steps or lemmas that are not yet fully correct. A proof structure may be interactively manipulated to refine or correct steps. To verify a proof structure, the infrastructure transforms the individual proof steps to prover-specific formats for external provers. The infrastructure makes sure that a proof structure is only marked as "verified" if all required proof steps have been successfully proven by an external prover. We present a concrete implementation of our design, called VeriTaS.

In summary, the contributions of this paper are:

- We propose the requirements of a configurable verification infrastructure that enables combining domain-specific problem specifications and proof strategies with different automated theorem provers and solvers (Section 2).
- We formally define the conceptual model for a verification infrastructure that satisfies the requirements we propose. Our verification infrastructure is based on generic proof graphs for structuring proofs (Section 3).
- We present a flexible Scala implementation of our conceptual model (Section 4), called VeriTaS. Our implementation allows for customization to domain-specific needs.
- We illustrate how the generic parts of our implementation can be instantiated for verifying properties of specifications that use algebraic datatypes and recursive functions (Section 5).
- We illustrate how this instantiation of our infrastructure can be used to develop a progress proof of a typed subset of SQL (Section 6).

2 REQUIREMENTS

Our goal is to design and implement a verification infrastructure which flexibly allows developers who want to automate proofs in a specific domain (e.g., type soundness proofs) to encode domain-specific proof strategies while at the same time enabling the use of existing ATPs or SMT solvers for solving individual steps within a proof. We start by deriving seven concrete requirements for such an infrastructure.

2.1 Decoupled Proof Construction and Step Verification

We want to automatically derive as much of a proof as possible with the domain knowledge that we have in mind for the chosen domain. In particular, we do not want our domain-specific proof strategies to get "stuck" at the first step that cannot be immediately verified. Rather, we would like to allow the generation of proof steps that are not yet fully correct or not yet fully refined, but can nevertheless provide a good base for generating further dependent steps.

We achieve this by requiring the verification infrastructure to strictly decouple the generation of a *proof structure* from the verification of the individual proof steps within the structure using an external prover. Hence, our infrastructure shall allow the generation of incorrect or incomplete proof steps (for which verification might of course fail).

2.2 Interactive Proof Manipulation

Due to our first requirement that the generation of a proof structure and the verification of its steps shall be decoupled from each other, we might end up with a proof that has steps that cannot be verified by an external prover, either because the step is not yet sufficiently refined, or simply because the step itself is wrong. To complete such a proof, we want to be able to interactively manipulate a generated proof, i.e. correct faulty steps or refine steps.

2.3 Structured Proofs

If we have to interactively correct a generated proof, we need to be able to inspect the entire proof in order to fix the issues. Hence, the verification infrastructure should allow for a structured, hierarchical representation of proofs. Furthermore, the representation of proofs should allow for an intuitive visualization of a proof structure.

2.4 Combination of different ATPs and SMT solvers

Different automated theorem provers and solvers have different strengths. For example, SMT solvers are typically very strong at solving large, mostly ground problems, while ATPs typically handle quantified problems more elegantly [2]. To obtain a high degree of automation, we want to be able to combine different ATPs and/or SMT solvers within one proof. This requirement is similar to one of the core concepts behind Isabelle Sledgehammer [3]. To meet this requirement, the verification infrastructure shall provide interfaces for connecting or calling various theorem provers.

2.5 Configurable Format for Specifications and Proof Obligations

Many domain-specific proof strategies (e.g. for lemma generation) may need to query domain concepts from the specification. If all problem specifications are given in a hard-coded general-purpose format, such queries may be difficult to implement for developers focusing on a particular domain: They would have to deal with

all the details of the AST of a general-purpose format to obtain the desired information for implementing domain-specific proof strategies. One way to get around such difficulties is to use a custom format for problem specifications in a particular domain which facilitates querying the information needed by domain-specific proof strategies. Such a format could contain special specification constructs that capture the domain knowledge required by particular strategies.

To allow developers for specifying custom formats, the verification infrastructure shall be *parametric* in a format for problem specifications as well as proof obligations.

2.6 Persistent Proofs and Verification States

When developing a proof, one usually wants to share either the finished proof or an intermediate state of the proof so that others can either inspect the proof to persuade themselves of its correctness, or complete the proof.

To this end, we require that proof structures as well as verification states of individual proof steps can be stored persistently. Storing of verification states should not only include a simple textual status of the form "proved" or "not proved", but also the prover used and its configuration (e.g. timeout, parameters...) that yielded this status, along with *evidence* for the proof. This allows other developers to complete inconclusive steps efficiently and to persuade themselves of the correctness of proved steps.

2.7 Expressive Language for Strategy Implementation

We want to be able to implement powerful and flexible proof strategies which, on the one hand, transform proof obligations, but, on the other hand, are also able to generate auxiliary lemmas needed within the proof by analyzing the present proof obligation as well as the problem specification. Hence, the language in which such strategies are implemented has to be expressive enough to allow for these kind of tasks.

We interpret this requirement on a technical level: We require the verification infrastructure to be completely implemented as a library within a general-purpose programming language, so that any proof strategy can also be implemented within this language. This way, the language for proof strategies automatically allows for the maximum flexibility and expressiveness. Also, developers do not need to work with different languages on different abstraction levels in order to implement proof strategies, but only with a single library within a single language.

3 CONCEPTUAL MODEL

To meet the requirements from the previous section, we represent proof structures via *proof graphs*, a format inspired by previous work on proof planning, notably on the work of Richardson and Bundy [17]. However, in our notion of proof graphs, we shift the focus away from tactic-based proof plans toward proof structures represented primarily by the intermediate subgoals that are generated during a proof. We discuss the differences to Richardson and Bundy's work further in Section 7.

3.1 Definition of Proof Graphs

We start with formally defining the core concepts and terminology of our notion of proof graphs. Section 6 shows an example of a visualized proof graph. The nodes of our proof graphs are *proof obligations*.

Definition 3.1 (Proof obligation). A *proof obligation* consists of a *problem specification*, i.e. a set of definitions and axioms together with a conjecture to prove. A proof obligation $p_{\mathcal{D},\mathcal{G}}$ is parametric in a format $\mathcal{D}$ for definitions and in a format $\mathcal{G}$ for proof goals/conjectures. We denote a set of proof obligations parametric in $\mathcal{D}$ and $\mathcal{G}$ with $\mathcal{P}_{\mathcal{D},\mathcal{G}}$.

Since proof obligations are parametric in a format for definitions $\mathcal{D}$ and in a format for proof goals $\mathcal{G}$, we meet requirement 2.5 (enabling a configurable format for specifications and proof obligations). To avoid notational clutter, we omit the subscripts $\mathcal{D}$ and $\mathcal{G}$ from now on and just write p (occasionally with numerical subscripts) for proof obligations and $\mathcal{P}$ for a set of proof obligations parametric in $\mathcal{D}$ and $\mathcal{G}$.

Proof obligations are connected to each other via directed *proof edges*.

Definition 3.2 (Proof edge). A *proof edge* is a triple $(p_1,\ l,\ p_2)$ from a proof obligation p_1 to another proof obligation p_2. A proof edge additionally carries an *edge label* l from a set of possible edge labels $\mathcal{L}$. We denote a set of proof edges labeled with elements of $\mathcal{L}$ with $\mathcal{E}_{\mathcal{L}}$.

Since proof edges are directed, a proof obligation can have children, or *sub-obligations*. The semantics of sub-obligations is that proving the parent obligation may require proving some or all of the sub-obligations, or, differently put, that the proof of the parent obligation may depend on proving the sub-obligations.

Edge labels may be used for different purposes, for example to propagate proof-relevant information from parent obligations to sub-obligations. Such proof-relevant information could include fixed variables or induction hypotheses from the parent obligation which also apply for proving the sub-obligations.

Proof obligations and proof edges serve as the basic ingredients of *proof graphs*.

Definition 3.3 (Proof graph). A *proof graph* is a directed, acyclic graph (DAG) $(\mathcal{P},\ \mathcal{E}_{\mathcal{L}})$, i.e. with nodes from a set of proof obligations $\mathcal{P}$ and edges from a set of labeled proof edges $\mathcal{E}_{\mathcal{L}}$. We denote the set of all proof graphs, i.e. the type of proof graphs, with *PG*.

Note that proof graphs have to be acyclic in order to represent correct proofs. We use graphs instead of trees in order to allow for proof obligations to be reused within a proof. For example, a sub-obligation may consist of an auxiliary lemma which is required in different parts of a proof. Rather than duplicating the proof obligation within a tree (including the sub-tree which models the proof of the auxiliary lemma), we allow for sub-obligations to have more than one parent obligation.

By representing proofs via proof graphs, we easily meet requirement 2.3 of structured proofs: Proof graphs enable a structured and hierarchical representation of proofs and can easily be visualized.

Proof graphs can have one or more *root obligations*, i.e. proof obligations without a parent obligation. Root obligations model the

theorems one wants to prove. *Leaf obligations* are proof obligations without any sub-obligation. A leaf obligation models a theorem or lemma which trivially follows from the definitions within the problem specification of the leaf obligation.

3.2 Constructing Proof Graphs

Next, we introduce concepts and terminology for the manual and/or automated construction of proof graphs. We may apply a *tactic* to a proof obligation in order to add a single *proof step* to that proof obligation.

Definition 3.4 (Tactic). A *tactic* t is a function of type

$$(\mathcal{P} \times \overline{\mathcal{E}_{\mathcal{L}}}) \to \overline{(\mathcal{P} \times \mathcal{E}_{\mathcal{L}})},$$

where $\mathcal{P}$ denotes the set of all proof obligations[1] and $\mathcal{E}_{\mathcal{L}}$ denotes the set of all labeled edges. We denote the set of all tactics with T.

That is, a tactic takes a proof obligation and a list of proof edges (the incoming edges of the given proof obligation) as an argument and returns a list of pairs of a proof obligation and a proof edge.

Applying a tactic to a proof obligation deterministically generates a list of sub-obligations with corresponding proof edges to each of the sub-obligations. The tactic requires the list of incoming edges to the given proof obligation in order to generate new proof edges and correctly propagate information if necessary. A tactic may also return an empty list. For example, the simplest tactic, which we name *Solve*, simply declares that a proof obligation can be proven directly via its problem specification. Hence, applying *Solve* does not generate any further proof obligations. Additionally, if the tactic is not applicable to the given parent obligation, the function t also returns the empty list.

Definition 3.5 (Proof step). A *proof step* s is a triple $(p,\ t,\ \overline{(p, e_{\mathcal{L}})})$. We denote a set of proof steps with $\mathcal{S}$.

The leftmost element of the triple represents the parent obligation p, the second element the tactic t applied to p, and the last element the list of sub-obligations with corresponding labeled proof edges $e_{\mathcal{L}}$ generated by t for p and its incoming edges. For each parent obligation p in a proof graph, there may be at most one proof step.

Applying a tactic t within a proof graph pg_1 to a parent obligation p generates a proof step for p and returns a proof graph pg_2 augmented by the proof edges and sub-obligations generated by applying t to p and its incoming edges, together with the generated proof step s; formally:

Definition 3.6 (Tactic application). The function *applyTac* of type

$$(PG \times \mathcal{P} \times T) \to (PG \times \mathcal{S})$$

is defined so that $applyTac(pg_1, p, t) = (pg_2, s)$, where

- $pg_1 = (\mathcal{P}_1, \mathcal{E}_{\mathcal{L},1})$
- $pg_2 = (\mathcal{P}_2, \mathcal{E}_{\mathcal{L},2})$
- $t(p, e_{\mathcal{L}}) = ((p_1, e_{\mathcal{L},1}), ..., (p_n, e_{\mathcal{L},n}))$
- $s = (p, t, ((p_1, e_{\mathcal{L},1}), ..., (p_n, e_{\mathcal{L},n}))\ )$
- $\mathcal{P}_2 = \mathcal{P}_1 \cup \{p_1, ..., p_n\}$
- $\mathcal{E}_{\mathcal{L},2} = \mathcal{E}_{\mathcal{L},1} \cup \{e_{\mathcal{L},1}, ..., e_{\mathcal{L},n}\}$,

if $p \in \mathcal{P}_1$. Otherwise, $pg_1 = pg_2$, i.e. the given proof graph remains unchanged.[2]

Tactics can only ever be used to construct a single proof step out of a given proof obligation. *Proof strategies* heuristically generate entire proof graphs:

Definition 3.7 (Proof strategy). A *proof strategy Str* is a function $PG \to PG$, i.e. a function from a proof graph to a proof graph.

Initially, a proof strategy may be applied to a proof graph that only consists of one or more root obligations without any sub-obligations. Proof strategies may apply other proof strategies and/or tactics in order to construct proof graphs. But most importantly, a proof strategy may operate *globally* on a proof graph and the problem specifications within its proof obligations, while tactics only ever operate *locally* on a given proof obligation. Thus, proof strategies may for example heuristically generate sub-obligations that contain auxiliary lemmas and insert them into certain points of the given proof graph.

Users may manually apply tactics as well as single proof strategies to refine a proof graph. Thereby, our model meets requirement 2.2 (enabling interactive proof manipulation).

3.3 Verifying Proof Graphs

The proof graphs constructed by proof strategies and/or via tactic application represent suggested proof structures and may contain unprovable proof obligations and incorrect proof steps. To actually verify a proof graph, each of its proof obligation has to be verified with an external verifier. We formally define what "verifying a proof obligation" means.

Firstly, to verify a proof obligation, one has to attach a proof step to the proof obligation via a tactic application. Next, an external verifier has to verify the attached proof step. A proof step is verified by first encoding the proof step as a *proof problem* in a verifier format $\mathcal{V}$ and then applying a *verifier* onto the proof problem in order to obtain a *step result*.

Definition 3.8 (Encoding proof problems). A function *enc* of type $\mathcal{S} \to \mathcal{V}$ encodes a proof step into a *proof problem* in a verification format $\mathcal{V}$.

Definition 3.9 (Step result). A *step result* is a triple $(s, stat, ev)$, where s is a proof step, *stat* is a proof status label that has one of the textual values {*Proved*, *Disproved*, *Inconclusive*}, and *ev* is any kind of *evidence* for the verifier's result. We denote a set of step results with $\mathcal{R}$.

The *evidence* for a proof result may, for instance, be a proof (for *Proved* results), a counterexample or contradiction (for *Disproved* results), or it can also be empty (e.g. for *Inconclusive* results). For example, some TPTP provers [21] produce proofs in the TSTP format [20], which we could use as evidence for *Proved* results.

Most importantly, we will use the given evidence for a proof to persist the current state of a proof as suggested in requirement 2.6.

Definition 3.10 (Verifier). A *verifier* is a function *ver* of type $\mathcal{V} \to \mathcal{R}$ that produces a step result $(s, stat, ev)$ when given a proof problem in a verification format $\mathcal{V}$.

[1] All proof obligations for a given format for definitions $\mathcal{D}$ and a given format for goals $\mathcal{G}$.

[2] An actual implementation of *applyTac* may return appropriate messages.

Definition 3.11 (Verifying a proof step). A proof step s is verified if, for a verifier *ver* and an encoding function *enc*,

$$ver(enc(s)) = (s, Proved, ev).$$

The verification of a proof step is only the first part of verifying a proof obligation. To fully verify a proof obligation, we recursively have to verify all its sub-obligations:

Definition 3.12 (Verifying a proof obligation). A proof obligation p in a proof graph $(\mathcal{P}, \mathcal{E}_{\mathcal{L}})$ (i.e. $p \in \mathcal{P}$) is verified if both of the conditions below hold:

(1) The unique proof step $s = (p,\ t,\ ((p_1, e_1), ..., (p_n, e_n))\ )$ is verified.
(2) All the sub-obligations of p are verified, i.e. $p_1,\ ...,\ p_n$.

A proof graph only represents a fully correct proof if all its proof obligations are verified. The proof graph itself only serves as a means to structure and assemble the individual proof problems within a proof. By making this distinction, we meet requirement 2.1 (decoupling proof construction and step verification). The individual proof problems represented by the proof steps may be verified by different ATPs and SMT solvers, as formulated in requirement 2.4.

4 IMPLEMENTATION

We implemented the configurable verification infrastructure that we suggest in Section 3 as a Scala [15] library that we named VeriTaS. Our implementation is publicly available at https://github.com/stg-tud/type-pragmatics/tree/master/Veritas.

We chose Scala because it is well-suited for creating and programming with embedded DSLs. An embedded DSL can utilize the syntax and semantics of Scala to provide domain-specific constructs [10]. This way, our users can create and employ domain-specific formats for problem specifications and obligations within VeriTaS. And of course, developers of domain-specific proof strategies benefit from the full expressivity of the Scala language to query specification ASTs and to program strategies. Hence, VeriTaS meets requirement 2.7 (expressive language for strategy implementation).

Figure 1 summarizes the main Scala traits[3] of our core API, listing the most important API methods for illustration. We also implemented an example instantiation of the traits shown in Figure 1, which we describe in Subsection 4.2. All components of the core API are parametric over a format for definitions (type parameter `Def`) and over a format for proof obligations (type parameter `Goal`), as formalized in Section 3. The main components of VeriTaS are the two traits `IProofGraph` and `ProofGraph`.

The trait `IProofGraph` (= *immutable* proof graph) defines the components of a proof graph and groups all read-only methods on proof graphs. The trait defines type members for proof steps, proof obligations, and step results. Each of these type members extends a corresponding generic trait (prefix `Gen`), which defines a few basic, minimal fields for proof steps, obligations, and step results. Concrete implementations of trait `IProofGraph` may provide custom data structures to instantiate the type members of `IProofGraph`. We will see example instantiations in Subsection 4.2.

[3]For readers not familiar with Scala: Traits are similar to Java's interfaces.

One of our design conventions is that all main obligations of a proof, and especially all root obligations, are stored with a string name. One can access a map of all stored obligations and their names via method `storedObligations` and access a single obligation via its string name using method `findObligation`. Given an obligation, one can access its proof step via method `appliedStep`, if the obligation has a proof step attached. Given a proof step, one can access the attached edge labels and sub-obligations via method `requiredObls`. Methods `isStepVerified` and `isOblVerified` allow for querying whether a proof step resp. a proof obligation in a proof graph is verified or not. Trait `IProofGraph` also contains further API methods which we omitted in Figure 1, for example methods which allow for accessing the parents of an obligation.

Trait `ProofGraph` extends `IProofGraph` with methods for modifying a proof graph. One may add a new proof obligation to a proof graph via method `storeObligation`, assigning a custom name to the obligation object. Most importantly, one can grow a proof graph by applying a `Tactic` to one of the graph's `Obligations` (method `applyTactic`). This will add a proof step and sub-obligations to the graph, but it will not yet verify that step. This way, we implement requirement 2.1 of decoupling proof construction and step verification: Proof strategies can construct the entire structure of a proof before any potentially unsuccessful verification is started. Thus, one may program proof strategies that do not get immediately stuck if a single verification step fails along the way. The verification of a proof obligation and potential correction of a generated proof graph is prover-specific and may require user intervention.

One can verify a proof step via method `verifyProofStep`. This method takes the proof step to verify and a verifier as arguments. A verifier has to implement trait `Verifier`, which has a type parameter `VerifierFormat` that can be instantiated as needed for a specific verifier. Trait `Verifier` consists of a method `verify` that has to produce a step result, i.e. an instance of `GenStepResult`. `GenStepResult` contains a field `evidence` that the verifier has to deliver. The format for evidence is prover-specific and so are evidence checkers. We store the evidence for each proof step in the graph, such that it becomes possible to check proofs developed by others. A concrete implementation of `Verifier` may for example consist of a custom verifier implemented within Scala, or call existing external ATPs and SMT solvers, translating their results into an instance of `GenStepResult`. We will see examples of concrete `Verifier` instantiations in Subsection 5.

Method `verifyProofStep` in trait `ProofGraph` calls method `verifyStep` within the `Tactic` instance stored in the given proof step. Method `verifyProofStep` passes the given verifier on to `verifyStep`, along with the parent edges that may contain information which could be needed for verifying the proof step, such as induction hypotheses or fixed variables. Method `verifyStep` first translates a proof problem in format `Def` and `Goal` to the `VerifierFormat` of the given verifier, then finally calls method `verify` of the given verifier and passes the translated problem.

4.1 ProofGraph extensions

To augment the proof graph core API with additional functionality, we created further utility traits and classes: First, we created a trait `ProofGraphVisualizer` for visualizing a given proof

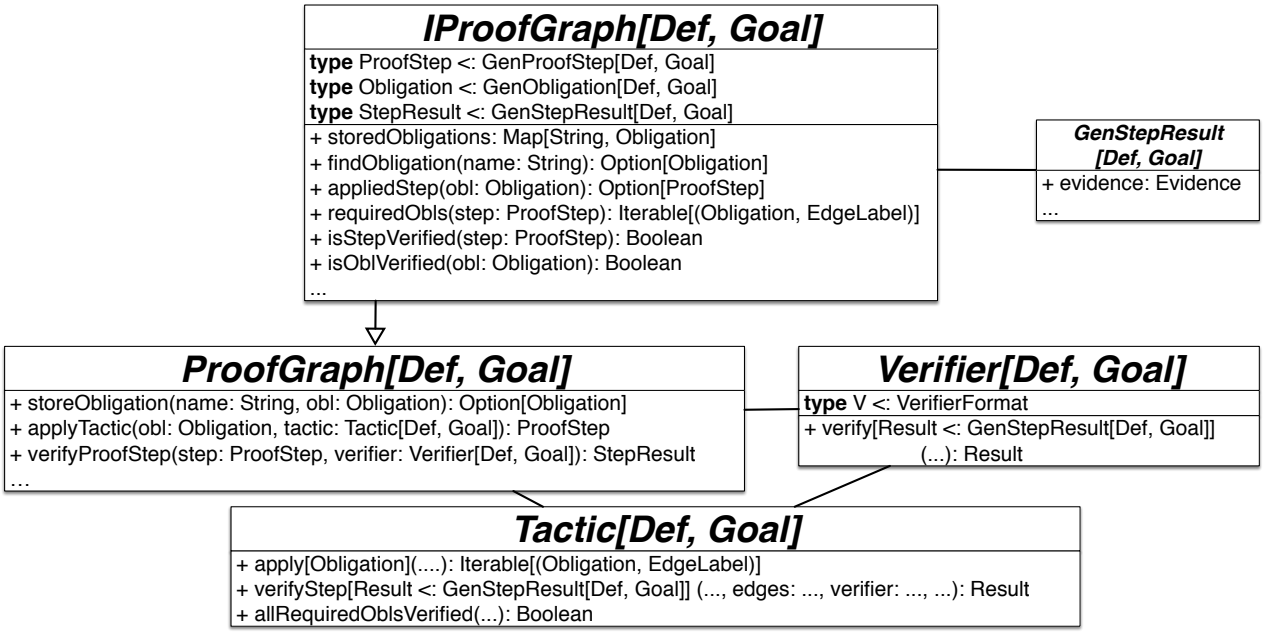

Figure 1: Overview of core API for proof graphs within VeriTaS

graph. This trait contains methods for encoding obligations, proof steps, and edges into a format that can be visualized. As an example, we implemented trait `ProofGraphVisualizer` via class `GraphVizVisualizer`, which translates a given proof graph into the dot format so that the graph can be visualized using GraphViz.[4]

Second, we created a trait `ProofGraphTraversals`, which extends `ProofGraph` with additional traversal methods for traversing obligations and proof steps, as well as for applying fold and map operations on obligations etc.

Third, we created a class `ProofGraphUI`, which allows for augmenting more convenient access of inner sub-obligations of a proof graph: One can pass a function when instantiating `ProofGraphUI` which calculates a string for a given obligation, and then one can access obligations within a given proof graph via the calculated name.

4.2 Reference Implementation of ProofGraph

We implemented trait `ProofGraph` using the Java database Xodus[5] for persisting proof graphs. Xodus is a transactional schema-less databases developed by JetBrains: The class `ProofGraphXodus` implements obligations, proof steps, and step results as entities within a Xodus database with links among each other. The fields of obligations, proof steps, and step results either become properties of the corresponding entities, or separate entities. For example, the goal of an obligation becomes a property of an obligation entity. However, we create separate entities for specifications, which in turn save the actual specification within a property. An obligation entity then links to a specification entity within the database. This has the advantage that we can link to one specification entity from many different obligation entities and hence improve on the overall size of the database: It is likely that for many problems, specifications will be large and that many obligations within a single proof graph will have the same specification. Hence, it makes sense to let several or even all obligations share a specification internally.

We implemented the methods inherited from trait `IProofGraph` via read-only transactions and the additional methods from trait `ProofGraph` via regular transactions. By using transactions, we ensure that edits of a proof graph do not lead to an inconsistent state, even processing a proof graph concurrently.

5 INSTANTIATING VERITAS

To illustrate the flexibility of our verification infrastructure, we present possible instantiations of the configurable parts of VeriTaS, namely a basic embedded DSL for problem specifications and proof obligations, the implementation of some standard tactics, and different concrete verifier instances that connect several existing ATPs and SMT solvers to our infrastructure.

5.1 Specification DSL

We implemented an embedded DSL in Scala that allows for specifying algebraic datatypes (ADTs), recursive functions, and properties in first-order logic as well as inference rules. Concretely, we first created a number of Scala classes for representing the abstract syntax trees (ASTs) of such specifications. All of these classes have a common superclass. To construct proof graphs for proving properties specified in our embedded DSL, we simply instantiate the type parameters `Def` and `Goal` of an implementation of `ProofGraph` (e.g. class `ProofGraphXodus`) with this superclass.

We use Scala's implicits for introducing a more comfortable syntax for using our AST classes: We implemented implicit methods

[4] http://www.graphviz.org/
[5] http://jetbrains.github.io/xodus/

```
object ExampleSpecification {
// some import declarations for making DSL
// constructs available omitted here
val NatType =
data('nat) of
'zero |
'succ ('nat)

val plusfunction =
function ('plus.>>('nat, 'nat) -> 'nat) where
('plus('x, 'zero) := 'x) |
('plus('x, 'succ('y)) := 'succ('plus('x,'y)))

val commutativity =
('plus('x, 'y) === 'z)
===>("addition-commutative")
('plus('y, 'x) === 'z)
}
```

Figure 2: An example specification in our embedded specification DSL.

and implicit classes which have the names of the operands we want to provide, and which convert the given arguments into the correct AST class.

Figure 2 shows a small example specification in our embedded DSL: Natural numbers and a recursive function for addition, together with the property "addition is commutative", given in an inference rule notation whose semantics is that all free variables are implicitly universally quantified. Here, operands such as "of, >>, – >, ===, ===>" are methods within implicit classes which return instances of the corresponding AST classes.

With this small specification DSL, we may already generate a number of very interesting specifications and prove properties on them. For example, we may use the format to specify the syntax, reduction semantics, and type systems of small DSLs (see Section 6). Instead of the specification DSL we presented here, we can easily generate other embedded DSLs for specifying proof problems and instantiate `ProofGraph` with them. One could for example create DSLs for specifying state automata, hardware primitives, cryptographic protocols etc. Developers may also create custom formats with for example domain-specific annotations, which then can be used to extract certain domain-specific information from specifications.

5.2 Implementing Tactics

We implemented a number of standard proof tactics with which one can prove simple properties given in our specification DSL. When implementing tactics in our verification infrastructure, the usage of a modern object-oriented general-purpose programming language allows us to employ any kind of advantageous software engineering techniques, for example in order to increase the reusability of the implemented tactics.

To illustrate how this may look like, we first defined a lightweight trait `SpecEnquirer` for querying specific information from problem specifications. The trait is, like all the other parts of our core API described before, parametric in a format for problem specifications and proof obligations. Trait `SpecEnquirer` contains methods for querying basic information from a specification format, such as for example "is a given proof obligation universally quantified?", "does a given variable in a given term have the type of a closed ADT?" etc. Additionally, it contains constructor methods for building proof obligations.

Next, we implemented basic tactics such as structural induction and case distinction as generic tactics that are again parametric in a format for problem specifications (`Def`) and in a format for proof obligations (`Goal`). The implemented tactics make no assumptions about how a specification format looks like, but employ solely the query methods from trait `SpecEnquirer` to obtain relevant information from parts of the given specification and the constructor methods of `SpecEnquirer` to build sub-obligations and proof edges.

Finally, we implemented trait `SpecEnquirer` for our specification DSL from Section 5.1.

This design allows the tactics we implemented to be reused with other specification formats beyond the one we showed in Section 5.1. To reuse the tactics with a custom format, developers only have to implement the `SpecEnquirer` trait for their own format.

Note that, in accordance with our requirement 2.1 of decoupled proof construction and step verification, tactic applications on proof graphs need not necessarily represent correct proof steps. A tactic only has to create a proof problem that can be passed to an external verifier, which then has to attempt the actual verification of the proof step. For example, the tactic for structural induction creates base cases, step cases, and induction hypotheses for a given proof obligation and a given induction variable, based on the type of this variable. The associated proof step consists of the induction cases and hypotheses and the parent obligation. To verify such a step, a verifier has to confirm that the generated induction cases conform to a valid induction scheme.

5.3 Connecting ATPs and SMT solvers

To connect different automated theorem provers (ATPs) and SMT solvers to VeriTaS, we need to

(1) implement a translation from the specification format for definitions and proof obligations that we use to an input format supported by the prover/solver we want to connect,
(2) parse the output or logs of the prover and translate them into the `StepResult` used by the `ProofGraph` instantiation that we use,
(3) implement the `Verifier` trait with a verifier that calls the external prover with the translated problem and parses its output.

We implemented different translations from our custom specification DSL into different TPTP dialects [21], which is supported by many different ATPs, and also into the SMT-LIB format[6], which allows us to connect SMT solvers such as Z3 [4]. Concretely, we implemented a translation pipeline which gradually translates the higher-level language constructs from our specification DSL such

[6] http://smtlib.cs.uiowa.edu/

as inference rules and abstract datatypes to lower-level language constructs, i.e. axioms with explicitly universally quantified variables. We describe different variants of this translation process in detail in a previous publication [7]. Ultimately, the last step of our translation pipeline simply translates the intermediate version of the problem specification that uses only lower-level language constructs into the target format (TPTP dialects FOF and TFF, or SMT-LIB).

We implemented `Verifiers` which call different versions of Vampire [11], and `Verifiers` which call Eprover [19], and princess [18]. We parse the results of these provers into `StepResults`. Vampire hands back TSTP [20] proofs, which we use as prover evidence.

6 CASE STUDY

We use our embedded specification DSL from Section 5.1 for modeling a typed subset of SQL that includes projection on columns, selection of rows based on simple predicates, and constructing the union, intersection, and difference of tables. We specify a small-step reduction semantics for queries, which reduces nested queries to simpler queries and ultimately to table values. We define a simple type system which checks that an SQL query adheres to the typed schemas of the tables it refers to, i.e. that column projections only project on existing columns, that predicates for row selection always receive values of the expected type, and that union/intersection/difference is only applied to tables with the same table schema.

We use VeriTaS to prove a *progress property* for our type system. Progress is part of proving type soundness [16]. Intuitively, progress demands that a correctly typed program expression (in our case, an SQL query) either already is a value (in our case, a table value), or can be reduced at least one step further by the reduction semantics.

Figure 3 shows part of the visualized proof graph of an intermediate state of the progress proof: We visualize proof obligations with rectangles, and the proof steps with diamonds. Proof steps contain as labels the tactics applied within the steps. The edges are labeled with case names generated by the tactic applications. The verification state of each proof obligation and each proof step is marked by color: Verified proof obligations and proof steps are marked in green, inconclusive ones in red.

We obtain the intermediate proof state from Figure 3 by applying Vampire 4.1 in CASC mode with a timeout of 120 seconds to all proof steps resulting from tactic applications except for the ones from `StructuralInduction` applications. For verifying the structural induction steps, we employed a small verifier which produces the used induction scheme as evidence for user inspection. This is just for illustration purposes - in principle, we could for instance attach a general-purpose interactive theorem prover such as Isabelle/HOL [1], e.g. via the Scala library libisabelle[7] and ask it to verify that our induction step indeed uses a valid induction scheme.

The root of the proof graph, `SQLProgress`, contains the progress property we want to prove. We first apply the `StructuralInduction` tactic, which generates the induction cases (all labeled `sub`, with labeled proof edges that contain generated case names). The leftmost induction case (`SQLProgressicase0`) represents the first base case, the case in which a query already is a table value and for which progress hence trivially holds. This case is easily solved by Vampire 4.1.

The next case, `SQLProgressicase1`, represents the second base case: A typical SELECT _ FROM _ WHERE _ query. Note that in our subset of SQL, we model neither joins nor nested SELECT _ FROM _ WHERE _ queries, hence this case represents a base case. Nevertheless, it is a very complicated base case, since these queries potentially modify the input table a lot: They first look up a given table in the table store, then select rows for which a given predicate holds, and finally project on the given columns. Each of these steps may fail, causing the reduction of the query to become stuck. Hence, we need a number of auxiliary lemmas to prove that the individual selection/projection steps will not get stuck for a well-typed query. Part of the corresponding sub-proof graph is shown in Figure 3 below `SQLProgressicase1`. As we can see, Vampire 4.1 proves most of the basic proof steps (the steps marked with the `Solve` tactic) and fails for some of the complex steps, e.g. for the application of four auxiliary lemmas (`LemmaApplication`) at the top of the proof. The remaining induction cases (of which only one is shown in Figure 3, namely `SQLProgressicase2`) refer to the cases of union, intersection, and difference of tables. For these queries, we allow nesting in our subset, hence these cases contain sub-cases which require applying the induction hypotheses, which is propagated along the edges of the proof graph.

This example firstly illustrates that it is beneficial to combine different automated verifiers with different abilities within a complex proof. Secondly, it illustrates how proof graphs help with visualizing a structured proof and locating the points within a proof where the applied automated verifiers fail. Next, we may either try to apply different verifiers to these proof steps, or refine the steps within the proof graph until the used verifiers find proofs for them.

So far, we constructed the current proof graph by manually applying the basic tactics from Section 5.2. However, our verification infrastructure would allow us to implement flexible proof strategies which exploit domain concepts for proof construction and lemma generation. In this paper, we focus on the design of a verification infrastructure which provides the means for implementing such strategies. The actual implementation of such a strategy is future work.

7 RELATED WORK

We first compare our verification infrastructure against existing interactive theorem provers. In the interactive theorem prover Coq [6], tactics for constructing proofs can either be written in OCaml, in the internal tactic language ltac [5], or in the dependently-typed and more recent internal tactic language Mtac [23] (a monad for typed tactic programming in Coq). In the interactive theorem prover Isabelle [14], tactics for constructing proofs can either be written in Isabelle/ML, or via a recent collection of tools for a "proof method language" called Eisbach [13], which allows for defining proof methods via Isabelle's Isar syntax [22]. Isabelle Sledgehammer [3] allows for using ATPs and SMT solvers within Isabelle for discharging subgoals. For Dafny [12], which is a programming language and program verifier for the .NET platform, there is a tactic language called Tacny [9].

[7] https://lars.hupel.info/libisabelle/tutorial.html

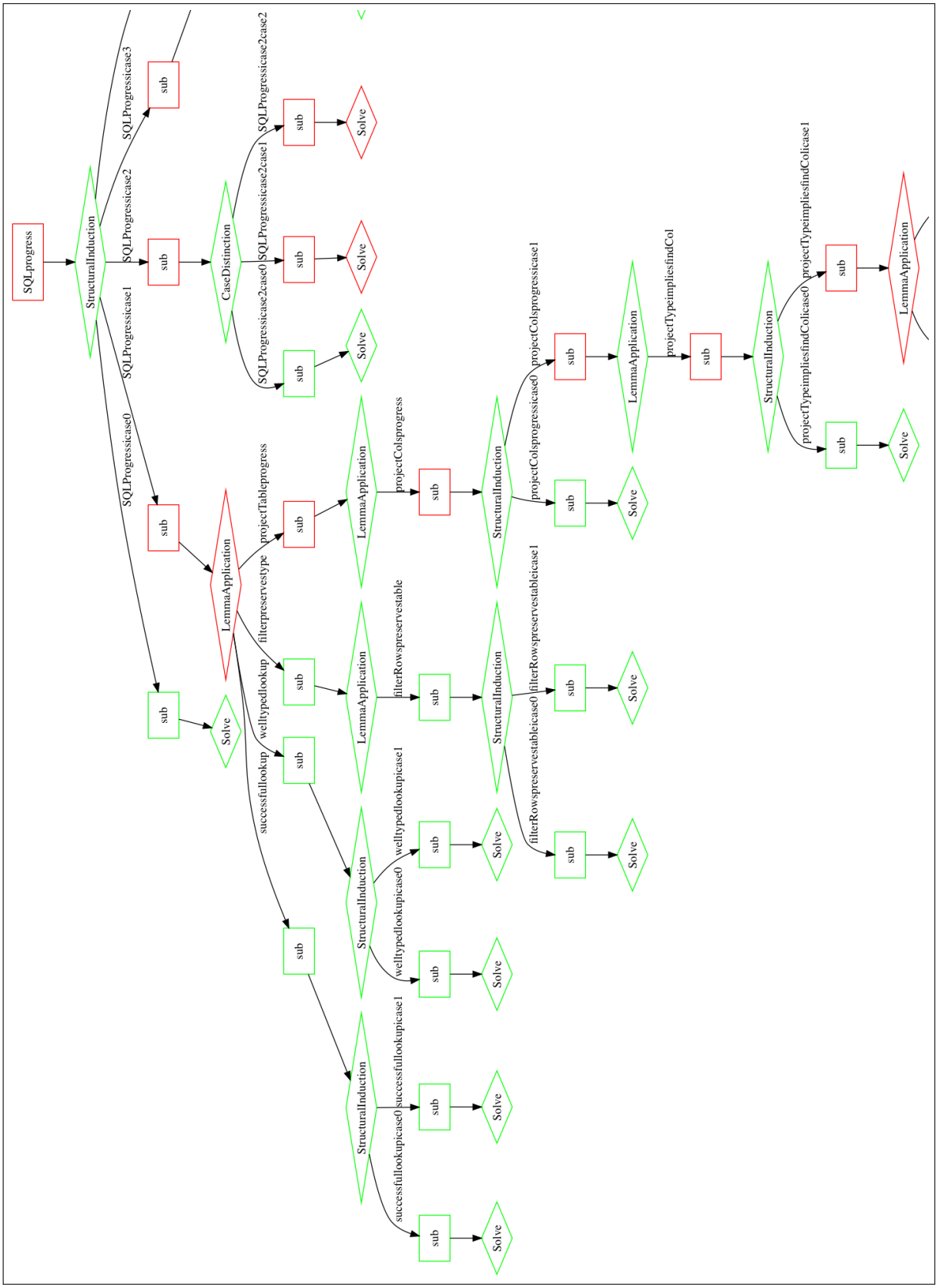

Figure 3: Excerpt of the proof graph of an intermediate state of the progress proof for a typed subset of SQL.

While the tactic languages just mentioned differ in how one can express and combine tactics and in which higher-order syntax constructs one may use for programming a tactic, they all have in common that they only allow for inspecting and querying the current goal state within a proof, and then manipulate that state by applying other available tactics to it. In general, tactic languages do not allow for querying the AST of a problem specification in order to for example inspect the different cases of a function definition.

Most importantly, existing tactic languages do not allow for the approximate construction of subgoals or auxiliary lemmas: Any intermediate goal can only ever arise from the successful application of the tactics from before. That means existing tactic languages cannot lay out an approximate *proof structure*, but only provide a plan for the *steps* to be executed to prove a goal. In the verification infrastructure that we propose in this paper, we deliberately take a different view on proof automation: We focus on representing a proof structure by explicitly forcing the generation of approximate subgoals and putting them into the center of how we represent a proof structure. The steps that have to be taken to get from one generated subgoal to another link these subgoals. We believe that this shift in focus enables a different, structural form of proof automation from which proof domains with very structured proofs such as type soundness proofs may benefit substantially.

Next, we compare the concept of proof graphs upon which our verification infrastructure is based to other graphical approaches for proof construction. We got the inspiration for proof graphs from the concept of *proof planning* by Richardson and Bundy [17]. Notably, in their work, Richardson and Bundy distinguish between the *meta-level logic*, i.e. the (possibly heuristic) logic used for constructing a proof plan, and the *object-level logic*, i.e. the formal system in which the actual proof is constructed. Reasoning at the meta level does not need to be sound, but reasoning at the object level needs to be sound. This corresponds to our requirement of decoupling proof construction and step verification (see Section 3). However, proof plans (which may be visualized via executable directed graphs, see for example the work of Grov et al. [8] on the Tinker tool) again consist of tactic nodes and thus are similar to tactic languages: Proof plans get stuck if a tactic gets stuck on the way, making it impossible to inspect the remaining hypothetical proof beyond the failure. In contrast, we base our proof graphs on intermediate proof obligations as nodes, which may be constructed as well as inspected even if a proof step further up in the proof graph is not verifiable.

8 CONCLUSION

We presented the design and implementation of a verification infrastructure that facilitates the automation of proofs by combining domain-specific proof strategies with automated theorem provers and SMT solvers. Our infrastructure allows for organizing proofs in *proof graphs*, whose nodes form the (intermediate) proof obligations that are connected via *proof steps*. The proof steps form independent, individual proof problems that can be passed to automated theorem provers and SMT solvers for verification. Most importantly, our infrastructure decouples the construction of a proof structure from the verification of its proof steps, which we believe is crucial for facilitating the implementation of domain-specific proof strategies: Within our infrastructure, such strategies do not need to construct only fully correct sub-obligations and proof steps, but may approximate steps. Good approximate steps may inspire proof developers towards finding the actually needed proof step, and hence are potentially more useful than missing steps. Nevertheless, we achieve soundness by making sure that a root obligation in the proof graph will only be marked as fully verified as soon as all dependent proof steps have been verified by external provers.

In the future, we plan to apply our verification infrastructure for automatically generating type soundness proofs of type systems for DSLs. We believe that the concepts presented in this paper will be useful in the future for automating proofs in different domains.

REFERENCES

[1] 2014. Isabelle documentation. http://isabelle.in.tum.de/documentation.html.
[2] Jasmin Christian Blanchette. 2012. *Automatic proofs and refutations for higher-order logic.* Ph.D. Dissertation. Technical University Munich.
[3] Jasmin C. Blanchette and Lawrence C. Paulson. 2016. *Hammering Away - A User's Guide to Sledgehammer for Isabelle/HOL.* Technical Report. http://isabelle.in.tum.de/dist/doc/sledgehammer.pdf
[4] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 337–340.
[5] David Delahaye. 2000. A Tactic Language for the System Coq. In *Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. 85–95.
[6] Cop development team. 2014. The Coq proof assistant reference manual.
[7] Sylvia Grewe, Sebastian Erdweg, Michael Raulf, and Mira Mezini. 2016. Exploration of language specifications by compilation to first-order logic. In *Proceedings of International Symposium on Principles and Practice of Declarative Programming (PPDP)*. 104–117.
[8] Gudmund Grov, Aleks Kissinger, and Yuhui Lin. 2013. A Graphical Language for Proof Strategies. In *Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. 324–339.
[9] Gudmund Grov and Vytautas Tumas. 2016. Tactics for the Dafny Program Verifier. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 36–53.
[10] Paul Hudak. 1996. Building Domain-Specific Embedded Languages. *ACM Comput. Surv.* 28, 4es (1996), 196.
[11] Laura Kovács and Andrei Voronkov. 2013. First-Order Theorem Proving and Vampire. In *Proceedings of International Conference on Computer Aided Verification (CAV)*. Springer, 1–35.
[12] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. 348–370.
[13] Daniel Matichuk, Toby C. Murray, and Makarius Wenzel. 2016. Eisbach: A Proof Method Language for Isabelle. *J. Autom. Reasoning* 56, 3 (2016), 261–282.
[14] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: A Proof Assistant for Higher-order Logic.* Springer-Verlag, Berlin, Heidelberg.
[15] Martin Odersky, Lex Spoon, and Bill Venners. 2011. *Programming in Scala: A Comprehensive Step-by-Step Guide, 2Nd Edition* (2nd ed.). Artima Incorporation, USA.
[16] Benjamin C. Pierce. 2002. *Types and programming languages.* MIT press.
[17] Julian Richardson and Alan Bundy. 1999. Proof planning methods as schemas. *J. Symbolic Computation* 11 (1999), 1–000.
[18] Philipp Rümmer. 2008. A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In *Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. Springer, 274–289.
[19] Stephan Schulz. 2013. System Description: E 1.8. In *Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) (LNCS)*, Vol. 8312. Springer, 735–743.
[20] Geoff Sutcliffe. 2010. The TPTP World - Infrastructure for Automated Reasoning. In *Proceedings of the 16th International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer-Verlag, 1–12.
[21] Geoff Sutcliffe. 2017. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning* 59, 4 (2017), 483–502.
[22] Markus Wenzel. 2002. *Isabelle, Isar - a versatile environment for human readable formal proof documents.* Ph.D. Dissertation. Technical University Munich, Germany.
[23] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. 2013. Mtac: a monad for typed tactic programming in Coq. In *Proceedings of International Conference on Functional Programming (ICFP)*. 87–100.