A Survey of Interactive Theorem Proving Systems and Their Practical Aspects in Large-Scale Formal Verification

Fariba Khan
School of EECS
Oregon State University
khanfari@oregonstate.edu

Abstract

Interactive theorem proving (ITP) systems let human and machine work together to build machine-checked formal proofs. Different ITP systems has emerged following different formal foundations of mathematics with their own set of interesting features. These systems are now being widely used in formal verification of large-scale complex systems to increase their reliability, leading to equally large-scale proof development. Consequently, the focus of proof assistant research has been expanded to practical aspects of ITP systems to identify and address domain-specific challenges in large-scale development. This survey organizes existing research on these practical aspects, along with a brief overview of interactive theorem proving systems themselves. It also draws attention to significant challenges and open problems identified by the researchers across literature.

Keywords— Proof Assistants, Interactive Theorem Proving, Proof Engineering, Large Scale Verification, Formalization, Formal Verification, Proof Development, Proof Maintenance, Proof Refactoring, Proof Repair, Proof Robustness, Usability.

1 Introduction

In recent years, interactive theorem proving (ITP) that assists humans in the intriguing task of building formal machine-checked proofs, has been emerging in the formidable area of large-scale complex system formal verification, leading to new challenges in proof development and maintenance. In theorem proving, the goal of proofs is to validate the correctness of a mathematical statement. Any mathematical proof can be spelled out into simple steps that can be verified irrefutably, and this is the main idea behind the machine checking of proofs. Proofs need to be translated from informal language to formal language i.e. formal system, where each proof step is either an axiom of the formal system or can be inferred from one through its valid inference rules. However, in the interactive arrangement, the machine's role is not only to check whether a human written proof i.e., proof script, is valid in some formal system but, more importantly, to assist in finding that valid formal proof. Different formal systems have prompted the development of different interactive theorem proving systems, also known as proof assistants, who play this role of machine. However, formalization of a system is different from the formalization in mathematics in the sense that it entails more than just the aforementioned translation. One needs to identify the system's properties that ensure its correctness before going about translating them into formal statements and finding corresponding proofs. Intricacy and scale of the set of properties reflect the complexity and size of the underlying system, and specification of these properties often evolves with the system as well. With the increasing formal verification of the large-scale complex system in proof assistants following successful ventures like verified C compiler CompCert C [1], proof development is now getting scaled at an unprecedented rate.

Therefore, proof development too calls for methodologies to facilitate efficiency and maintainability i.e., proof engineering, just like efficient software development demands for software engineering. These proof engineering aspects of proof development in ITP systems are considered as practical aspects of proof assistants in this survey. Some impressive works have already been done by researchers to identify challenges in large-scale system verification along with efficient methodologies to handle them. Still, proof engineering is far away from reaching the maturity level of software engineering. This survey presents existing research on practical aspects of ITP systems in large-scale system verification by categorizing them with respect to the proof engineering concerns they address i.e., identifying the challenges and requirements in large-scale system verification (Section 3), providing methodologies for efficient development of maintainable proofs (Section 4), aiding in maintenance of existing proofs over time through automated techniques (Section 5), supporting development through tooling and Integrated Development Environment (IDE) from ITP systems (Section 6). However, before diving into research on practical aspects, this survey provides an overview of the ITP systems to acquaint readers with salient characteristics of different systems (Section 2). Throughout the survey, it also points out the difference among ITP systems both in terms of their features and their practical aspects, as well as draws attention to significant challenges and open problems identified by the researchers.

2 Overview of Interaction Theorem Proving (ITP) Systems

This section provides a brief overview of the ITP systems in terms of their major distinguishing characteristics followed by a detailed comparison of a small set of representative systems. Although the focus of this survey is not on the formal logic underlying the proof assistants, to understand the difference among them as well as the techniques from the literature described in later sections, we need to talk about theories at least to some extent. In fact, salient characteristics that distinguish different ITP systems boost from the formal foundations they are based on. In the following, these ITP systems are introduced as they historically emerged following the development of the different formal foundations of mathematics informing readers to both simultaneously.

The first attempt in mechanical verification of mathematical proofs is the Automath [2] system, initiated by De Bruijn in 1967. It treats proof as a first-class object like any other term in formal language. This technique, in logic, is known as Curry-Howard-Isomorphism that states one-to-one correspondence between formal proof calculi and type systems for a model of computation, e.g., between natural deduction and typed system for lambda calculus. Martin-Löf's Intuitionistic Theory of Types [3] extends the Curry-Howard-Isomorphism to predicate logic by introducing dependent types and then later, after the discovery of Girad's paradox, replaced the original impredicative version with a consistent predicative version. There are both extensional and intensional variants of this type theory, which are implemented in proof assistants Nuprl and Agda, respectively. I'll explain both predicativity and intensionality later in the section.

On the other hand, under Curry-Howard-Isomorphism, Girad's polymorphic λ-calculus, system F_ω, corresponds to higher-order logic. Theory Coquand, in 1985, combined system F_ω with impredicative Martin-Löf type theory in a higher-order extension of Automath language and presented Calculus of Constructions (CoC) [4]. CoC was later extended with primitive inductive extension leading to the Calculus of Inductive Constructions (CIC). The proof assistant, Coq, is the implementation of this calculus. The current Coq system's logic is dependently typed, higher-order, partially predicative, and intensional [5].

Automath also initiated the concept of Logical Framework. De Bruijn's systems only give basic mathematical mechanisms and let the user implement their preferred logical rules. Thus, in a logical framework, one can embed a logic as a signature in the framework's type theory, and the provability of a formula in the embedded logic comes down to the type inhabitation problem of the framework's type theory. Systems LF, Twelf are some implementations of Logical framework. Around the same time as Automath, another deductive system, the logic of computable functions was devised by Dana Scott [6] and later, implemented in an ITP system, called Logic for Computable Functions (LCF), by Robin Milner and his collaborators at Stanford, Edinburgh, and Cambridge. Milner later developed the *LCF approach* for his systems to not to require storing and rechecking of large proof objects [7]. This LCF approach provides soundness by construction by making proofs terms of an abstract data type *thm* whose predefined values are axioms and constructors are primitive inference rules of the logic. One can only create a term of *thm* via the inference rules; thus, the proof has to be sound. Several present ITP systems, including as Isabelle [8], HOL [9], and HOL-Light, are successors of LCF system. Isabelle, in fact, is an LCF style logical framework with the logic directly encoded in LCF style serving as meta-logic for other logic to be embedded as object logic. Examples of embedding are Isabelle/HOL and Isabelle/ZF, where HOL is classical higher-order object logic, and ZF stands for first-order Zermelo-Frankel set theory.

There are few other active ITP systems that have not been mentioned above. Among them, Mizar is quite notable for its largest repository of formalized mathematics called the Mizar Mathematical Library (MML). Mizar's other famous and influential feature is its declarative input language, which enables batch proof checking. It can check a whole file at once even if there are some proof steps not acceptable by the solver, consequently creating a syntactically correct proof with potential holes in it. The precise notion of this method can be found in Wiedijk's work [10]. This feature, named Mizar Mode, has been incorporated in many proof assistants, including HOL, HOL Light, Isabelle, Coq [7, 11]. A complete overview of all other ITP systems such as LEGO, Lean, Idris can be found in [12, 13, 7].

2.1 Detailed Comparison of a Few ITP Systems

For the detailed comparison, I have selected the representative systems Coq, Agda, Nuprl, Isabelle/HOL in the attempt to cover most formal systems and their branches. They are compared based on the following categories that have practical implications on proof development.

Equality The definition of *equality* in type theory has great implication on proof development and type checking in the corresponding proof assistant. Extensional and intensional type theories handle *equality* differently. Extensional equality considers two objects to be equal if their external properties are alike i.e., they behave the same way with its environment. Intensional equality, on the other hand, requires the internal structure of the entities to be equal as well. For example, $f(n) = 3*n+9$ and $g(n) = 3*(n+3)$ are equal in extensional sense but not, in intensional sense. Similarly, two functions are considered equal by extensionality, known as *function extensionality* if they produce same outputs for the same inputs, where in intensionality, their implementation needs to same as well. Therefore, in proof assistants, based on intensional type theory such as Coq, Agda, if inductive definition of plus (+) defines $0+n=n$ but not $n+0=n$, then any statement of the later construct needs additional induction reasoning to hold i.e. needs to be proved. On the other hand, in extensional type theory based proof assistants such as Nuprl,type checking is undecidable. The reason is in extensional type theory, *definitional equality* that refers to equality by construction, and *propositional equality*, used in propositional statements to be proved, coincide. Consequently, undecidable programs can be specified in the type system making type checking undecidable. However, in intensional type theory *definitional equality* and *propositional equality* are completely distinct, and type checking is decidable.

De Bruijn Criterion De Bruijn Criterion [14] requires proof assistants to produce an independently checkable proof object for the interactively created proof in the system. This proof object should be easily verified by skeptic users by writing a simple type checking algorithm. This is one of the four ways identified by H. Guevers [7] to guarantee the correctness of ITP systems. Coq, Agda, Nuprl all meet the De Bruijn criterion. However, systems based on LCF, such as Isabelle, do not produce a complete proof object as each step is constructed with the guarantee that it is correct. In that sense, they do not meet the *De Bruijn criterion* though it should not be hard to generate a proof object on the side if needed. The practical implication is that, in the LCF approach, there is no need to check the system with respect to the theory anymore as its inference rules and axioms are implemented directly to create the proof.

Predicativity Predicativity of type theory has impact on its consistency as well as on its expressiveness. Impredicative refers to self-referential definition i.e., a definition that mentions or quantifies over itself. To be predicative, a definition can not mention itself, and if it contains quantifiers, then they can not be initiated with an object defined by the definition. Impredicative type system can be shown inconsistent by Girad's paradox thus Martin-Löf made his type theory predicative. Agda, Nuprl, meta-logic in Isabelle, all are predicative. However, in Coq, Type universe, including Set is predicative, but the Prop universe is still impredicative due to the informally agreed notion that it is often easier to express mathematical proofs with impredicativity.

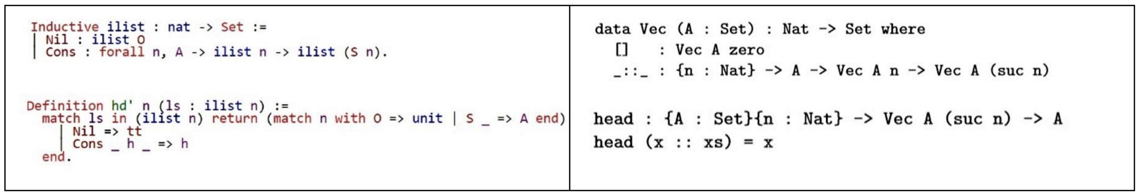
Termination and Totality In proof assistants based on intuitionistic type theories such as Coq, Agda, and Nuprl, functions need to be terminating for the logic to be consistent. Recursive calls have to be made on the subterm of the arguments to ensure their termination. For advanced cases where termination cannot be proven automatically, explicit proof of termination needs to be provided, or special approaches like exploiting the size of argument terms [15, 16] needs to be used. Agda provides induction-recursion that lets the user define mutually recursive functions and data types. Coq does not have this mechanism. However, classical higher-order logic in HOL, do not need a termination witness, but the use of the non-terminating function is limited [16]. Moreover, functions in HOL and CIC logic need to total, and partial function needs indirect encoding into the logic. These requirements are two difficulties beginners in ITP usually encounter, even if they are already familiar with functional programming language [16].

Proof Creation Style Proof creation style varies widely among ITP systems. In Coq, although it supports forward style, the usual way to create a proof is to use backward style (top-down). The user keeps applying tactics to the current goal, which creates subgoals needed to be proven separately, until True is reached. Proof development in Nuprl also follows the top-down approach. However, in Isabelle, the usual way is the forward style (bottom-up), where proofs are created from axioms and inference rules from the logic. Proof context gets enriched by new facts keeping the goal same, until the goal is reached [17]. However, both of them support tactics and provide tactic languages Ltac and Eisbach, respectively, to allow writing custom tactics. They also have proof languages like Coq/SSReflect and Isabelle/Isar. Proof languages aid in writing understandable proofs inspired by mathematical vernacular, in contrast to unstructured sequences of tactics. Users can reconstruct an English proof just form the script of the proof instead of stepping through tactics and tracking goal transformation.

However, in Agda, there is no support for the separate tactic language, although an automatic proof search tool, Agsy that works like *auto* tactic in Coq is there. Agda's dependent types let the user write almost any proposition. Proofs can be written as the term of the type of its theorem proposition. However, users can build proofs interactively in Emacs or atoms by putting a question mark (?) in the place of an expression, usually the proof term they need to write. Agda is able to type check incomplete programs with holes, i.e. (?) marks and can provide a list of the context, infer the type of the expression at the hole, and even sometimes evaluate the open term if bound variables in surrounding context are provided. Agda, like Idris, is basically a dependently typed programming language whose type storing and dependent types let it act as a proof assistant. The syntax in Agda is Haskell like where Coq's syntax is ML like.

Dependent Types and Pattern Matching on them Coq, Agda, and Nuprl support dependent types, but Isabelle/HOL does not have dependent types. However, it is easier to pattern match on dependent types in

Agda than in Coq. In coq, one needs to use complex clause like *match.....as.....return* clauses for that. Examples of pattern matching on indexed datatypes in Coq and Agda are shown side by side in the following Fig 1.



```
Inductive ilist : nat -> Set :=
| Nil : ilist O
| Cons : forall n, A -> ilist n -> ilist (S n).


Definition hd' n (ls : ilist n) :=
  match ls in (ilist n) return (match n with O => unit | S _ => A end)
    | Nil => tt
    | Cons _ h _ => h
  end.
```

```
data Vec (A : Set) : Nat -> Set where
  []    : Vec A zero
  _::_ : {n : Nat} -> A -> Vec A n -> Vec A (suc n)

head : {A : Set}{n : Nat} -> Vec A (suc n) -> A
head (x :: xs) = x
```

Figure 1: Pattern matching on indexed data-types in Coq and Agda respectively [18] [19]

Program Extraction Program extraction from proofs let certified program to be extracted and executed in favorable run-time environment.This mechanism is beneficial for system verification projects. Coq supports the extraction of executable programs in Ocaml and Haskell from proofs. Agda does not have any such support at this moment. Isabelle/HOL also has a code generation scheme based on the work of Haftmann and Nipkow [20]. It translates HOL to an intermediate language called Mini-Haskell and then, translates from this Mini-Haskell to Ocaml, Haskell or Standard ML.

Classical axioms Classical axioms cannot be proved in all proof assistants because of their underlying logic, thus require special techniques when they are needed in proof development but not supported by the ITP systems. Isabelle/HOL is based on classical higher-order logic. Hence, all classical axioms, e.g., Excluded middle, Double negations, are valid here. However, in intuitionistic logic, not all of them hold. For example, the double negation introduction law can be easily proven in intuitionistic logic, but neither the double negation elimination rule nor the Excluded middle can be proven. One way to embed Excluded middle axiom into intuitionistic logic is by double negating it, which is known as Glivenko's double-negation translation. Other classical axioms can also be translated through various translations such as Kudro's translation, Gödel-Gentzen translation [17]. Coq provides a library, Coq.Logic.Classical_Prop with classical axioms, to extend its intuitionistic logic to classical logic.

3 Challenges in Large-Scale Formal Verification

One important application of iterative theorem proving systems is formal verification of large complex systems. An increasing number of large scale system verification projects has been carried out in recent decades and has appeared as published case studies in literature. One of the successfully commercialized verification projects is the certified C compiler, CompCert [1, 21], for which Leory's work [1] received the Test of the Time award in POPL (Symposium on Principles of Programming Languages) 2016 and was noted as a groundbreaking contribution in demonstrating the feasibility of interactive theorem prover [22]. It's indeed a groundbreaking achievement even in terms of effort, given the scale of the project with 350,000 lines of Coq code along with state of the art of ITP systems at that time. Having tens of thousands of lines of code is not at all unusual in large proof development, irrespective of the ITP systems used in the project. Certification of seL4 Operating System Kernel by Klein *et. al.* [23] produced about 480,000 lines of code in Isabelle/HOL [24], Coq correctness proofs of an Raft distributed consensus protocol [25] comprises of 50k lines in total [26]. However, hurdles behind these verification works are not only due to the size. The biggest challenge comes from the complexity of the underlying system such as non-trivial concurrent OS kernel [27]. Industrial design specification is often performance-oriented, thus imprecise in nature, which impedes the creation of clear and elegant abstract specifications needed for formal verification [28]. Sometimes verification even gets carried out as part of an active, hence evolving development project. Interactive theorem proving is itself hard, and together with the issues mentioned above, it takes multiple persons years of effort for a single verification project [29]. Proof engineers and researchers involved in many of these large-scale projects later reported they could have saved lots of time if, from the beginning of proof development, they followed some schemes that facilitate adaptability, well-organization and reusability of proofs [30, 28, 26]. In fact, these attributes of proofs have been repeatedly identified by researchers that promote maintainability and efficiency i.e. the two main challenges in large scale development. Klein *et. al.* [24] also provides a picture of how often they needed to adapt their proofs to changes and how bad the maintenance cost was for that at the time of their sel4 OS kernel verification [23]. 17% of their total proof effort was just to reverify their system in response to fundamental changes in existing features of sel4 kernel. This is only one of the four kinds of changes they encountered. To be noted here, this verification project was carried out as part of the development of the kernel itself.

Not surprisingly, many of these large projects were followed up with work on methodologies for maintainable and efficient development. Although these methodologies were initially devised to be used in their respective projects, in the follow-up papers, they tried to generalize their solutions to be used irrespective of domains[30, 29, 28, 26]. Apart from these, some notable works have been done on proof reusability. However, even with conscious attention to maintainability during proof design and development, some unforeseen or unpredictable changes are bound to happen. Adapting proofs in large scale development to these changes has been identified as another challenging aspect of large-scale proof engineering. Researcher have also begun working on refactoring and repairing techniques to incorporate these changes efficiently. Therefore, research on maintainability and efficiency issues of large-scale proof development can be categorized as proactive measures to be followed during proof design and development and reactive measures to adapt proofs to changes anytime. Following two sections address them respectively.

4 Methodologies for Efficient and Maintainable Proof Development

Recent advancement in large-scale verification has expanded the focus of interactive theorem proving research from mere proof creation to practical proof engineering aspects [30]. Researchers are trying to come up with methodologies for designing well-organized and adaptable proofs to make development process efficient and maintainable [28, 26, 30]. The reusability of proofs also plays an essential role in promoting efficiency and maintainability. In the following two subsections, methodologies accumulated across literature are discussed under the two subcategories - a) methodologies for adaptability and organization, b) methodologies for reusability along with an account for how these strategies differ concerning ITP systems when applicable.

Moreover, proof engineers also benefit from some form of automation within the ITP as well as from the portability of proofs among different ITP systems to improve their performance, thereby efficiency of the overall proof development. A third subsection elaborates on gains from automation and portability as well as existing methods to facilitate them.

4.1 Methodologies for Proof Organization and Adaptability

Crafting robust, easily modifiable, strategically structured proofs play an important role in increasing efficiency during development. Features like modularity and abstractions in ITP systems, as well as custom or general-purpose tactics, can promote well organization and adaptability in proof development.

Through Modularity and Abstraction Modularity and abstraction can be used to write structured proofs. Organizing proof with good structuring principle often helps developers to compartmentalize their thoughts as well. For example, decomposing complex problems into simpler ones hierarchically based on their mutual dependencies lets the developer focus on one problem at a time. It also makes proving higher-level problems easier with already proved simpler ones they depend on. Software engineering has several methodologies for organizing and writing adaptable codes. Although these methods can give ideas for proof engineering, they often need to be improvised to be used in proof development. Information hiding from dependents or clients is one of these software development techniques, albeit a classic one. The trick here is one can not depend on the information they do not know, thus forcing dependents to write implementation without that information. This technique lets code to be updated without any changes to its clients. Proof engineering can also use this same technique by hiding functions' and types' definitions behind interfaces and reduce unnecessary rework. However, in proof development, dependents do not just use these definitions; they need to reason about them, which often demands more information. For instance, systems based on intensional type theory like CoQ depend on *definitional equality*, thus needs detailed constructional information of function and types as well. Planning for change by Woos *et. al.* [26] addresses this problem by adding lemmas in the interface that are sufficient and necessary to reason about themselves while hiding the rest of the information. To be noted here, systems based on extensional type theory, like Nuprl, would not need these additional interface lemmas when reasoning for equality (see section 2 for detail). They also recommend using the interface to separate theorem statements from their proofs. This technique also leads to faster development. In proof assistants like Coq, checking after modifying one proof kicks off rechecking of all other proofs that depend on it. Thus, the developer needs to wait a while after each edit, and in a large development, 'a while' gets really long. Following the above recommendation, dependent proofs would only have to import interfaces containing statements. Unless interfaces are not modified, the rechecking of other proofs is not needed anymore. Authors have reported the attainment of 100X faster build time by following this rule.

However, the realization of any of the above methods requires interfaces. Interfaces can be implemented through modules in proof assistants. CoQ module system, inspired by modules in standard ML, provides support to accumulate abstract types with operations over them [18]. Functors, another feature borrowed from ML, which are functions from modules to modules, are also supported in Coq that let create parametric modules. Coq also provides both options of either hiding or revealing implementation details when ascribing signatures. Chlipala in [31] has taken the advantage of the Coq module system to build a generic program verification tool with a library of reusable functors. This tool can be used to prototype verifiers for various type systems along with soundness certificate. Other example implementations of the module system in Coq are Finite sets and Maps using AVL trees [32], balanced binary search trees using red-black trees [33]. Agda also offers a module system. However, Agda modules can contain submodules along with definitions, thus allowing hierarchical organization within it. Functions defined in a module can be declared private to make them inaccessible from outside. Parameterization can be done through abstracting several arguments from multiple functions inside the module at once and making these arguments, parameters of the module instead. It is different from the use of functor in CoQ modules but analogous to sections in CoQ. Isabelle provides module system through extension called locales [34, 35] that supports general parameterization [36]. Rabe and Schürmann also, through a conservative extension, propose a module system for the logical framework for LF [37]. It has been implemented as part of the Twelf distribution and used to modularize big part of Twelf example library without any loss in efficiency. Type classes and canonical structures can also provide means for abstraction, and a detailed overview can be found in [16].

Even with all the module support from the system, structuring and decomposing code to achieve effective modular reasoning still require additional engineering and is a challenging open problem. Although lots of factors from the underlying system dictate the ultimate proof structure, there are some general principles that can be followed irrespective of that. One such general methodology can be found in Kaivola and Kohatsu's industrial-size circuit verification work [28]. Here, they structure their proof scripts by first layering the definitions hierarchically. Each definition layer captures a single aspect of the underlying system with associated claims. The higher-level definition can use lower levels' claims to reason about them. Then, they packaged all proofs that require internal details of a definition with the definition itself. Although there can be a proliferation of claims if definitions are refined too much, it still is a good technique to keep in mind when structuring proof. However, their working environment did not support modules, and they had to emulate them through conditional load sequences which incurred additional work. Using the custom induction principle is another way to achieve modularity [26]. Induction is one of the widely used methods of theorem proving in proof assistants with inductive data types. In these cases, common inductive arguments can be factored out into custom induction principles. Then, they can just be proved once and used throughout the system. Proofs that use induction principles remain valid after changes that maintain induction patterns and might just require reproving of the induction principles.

Refinement proof can also help in the organization. Abstract specification and low-level executable specification of a system can be related through refinement proofs in a way that if anything is true for the abstract specification, then it is true for implementation as well. Therefore, both specification and implementation can be kept and maintained separately. This can make low level changes like adding simple features less costly [24].

Through Tactics In tactic based theorems, use of general purposed tactics promote adaptability by making proof scripts robust to changes. These tactics can dispatch similar but different goals and proof scripts stay valid as long as modified goals can be handled by them. For example, *lia*, the newer version of the popular *omega* tactic, in CoQ can solve quantifier-free problems in Presburger Arithmetic. As a result, goals can be modified without breaking proof as long as they remain within the realm of *lia*'s solving capability. Custom tactics can also be written to facilitate robustness in tactic proofs. One of the recommendations in Planning For Change [26] for robust development, is not to depend on automatically generated hypothesis names or hypothesis ordering by systems. Any changes in the original goal is likely to modify these names and orders causing proofs to break if not followed the recommendation. One way to avoid this is to assign explicit names to hypotheses and the IDE Company-Coq has built-in support for that. However, these names also require maintenance in changing context. Instead, writing custom tactics, using tactic language like Ltac, to specify declaratively which hypothesis to be used rather than using names solve the problem more elegantly. For instance, the tactic *find_rewrite*, used by the authors in [26], can automatically find equality in the context and rewrites everywhere by it. StructTract library [38] has more custom tactics like this such as *break_match* and *prep_induction*.

4.2 Methodologies for Proof Reusability

Proof reuse can maximize the efficiency of developers by letting them use existing proved theorems and lemmas as much as possible, thereby reducing works when creating or adapting proofs. Several methods have been discussed across the literature to enable proof reuse. On a higher level, all these techniques are basically based on any one of the following three - a) proof term transformation or b) proof script generalization or c)exploiting type equivalence relationship.

Through Proof Term Transformation Proofs can be generalized by transforming associated proof terms in an ITP system whose underlying type theory represents theorems as types and proofs as terms of these types following Curry-Howard isomorphism. Examples of such ITP systems include Coq, Nuprl, Lego. Often times, proof development ends up with sets of specific theorems that are just specialized versions of general theorems. Hence, once a proof is proved for a specific theorem, one can generalize it to facilitate its later reuse. Related experimental work can be found in the Ph.D. thesis of Olivier Pons [39], which is, in fact, one of the earliest works on generalization through proof term transformation. The main idea in this thesis is to get an abstract version of the proof term by exploiting the dependency relationship of objects in the proof and then, use this in other places with appropriate instantiation. Another source of repetitive work in proof development is when the inductive types are extended with new constructors or new parameters. Developers again need to prove or adapt all the lemmas and theorems associated with the modified type. However, theorems on inductive types are usually proved by case analysis or by induction. Most cases of existing proofs can be reused without modification, and just the new cases require the developer's attention. Boite in his paper [40] proposed a tool that adapts existing proofs to these modifications by transforming associated proof terms. However, cases for new constructors are just automatically added in the adapted proof and developer needs to complete these cases by themselves. Mulhern's Proof Weaving [41] paper takes this process one step further. They proposed a method that attempts to automatically synthesize proofs for these new cases from the repetitive pattern of the existing proof structure. Consequently, the developer only needs to deal with non-repetitive interesting cases. LCF based proof assistants, by default, do not create an explicit proof term and provide soundness by construction. However, to be able to use techniques that rely on proof term transformation, Berghofer, and Nipkow [42] proposed proof term construction method for LCF and implemented it in the proof assistant Isabelle. Manipulation of these proof terms in Isabelle is also less involved than it is in proof assistants with richer type theory e.g., CoQ. This allows for more complex transformation such as both function and types abstraction without imposing any restriction on proofs themselves [40].

However, Caplan and Harandi [43] used a different approach in their work on the verification of reusable software components. They provide a logical framework by embedding a quantified version of Hoare logic into typed lambda calculus. This framework allows the simultaneous construction of abstract programs and its accompanying abstract proofs thus, enabling reuse of both through specialization.

Through Proof Script and Tactic Generalization Reusability can also be attained through abstraction at the level of proof scripts or tactics. Generalizing tactic based proofs allows reusing part of existing proofs after any changes to goal or specification. Felty and Howe [44] proposed an ITP system that generalizes its tactic proofs with metavariables after each tactic is applied. When a change is made, the reuse operation of this system solves a higher-order unification problem to find part of the existing proof that can be reused. Then it presents the adapted proof with remaining subgoals to the users. Their system is almost like Nuprl but with less restriction on inference justification to enable the use of metavariable that is not supported in Nuprl.

Moreover, some proof assistants provide features like modules and higher-order parameterization. These features facilitate separation of concerns and parameterization of proof specification, which could be used to promote reusability and maintainability in proof scripts. Delaware *et. al.* [29] exploited these in their work on verifying product lines of programming language. Product lines programs are decomposed into features, and variants of products are synthesized from composing different combinations of them. For composition, each feature uses variation points to capture the interaction between them. For example, the extension point of a data type construct in one feature when composing with another is marked by a variation point. On the verification side, authors also feature decomposed theorems into modules. Then, inside each feature module, variation points are encoded in theorems by higher-order parameterization. These modules now can be certified independently by appropriate assumptions on the variation points, thereby not requiring any recheck of existing proofs after compositions. Other formalization with similar needs like abstraction and modularity can use this technique in any higher-order proof assistant.

Through Type Equivalence In recent years, exploiting type equivalence relationships to facilitate reusability is seeing increasing popularity among proof engineering researchers. These relationships promote reuse by allowing functions and proofs, written for one type, to be carried over to an equivalent type. Earlier related works include Barthe and Pons' work on Type Isomorphism in dependent type theory [45]. They proposed an extension to dependent type theory by enforcing type isomorphism through rewrite rules at the level of types and elements. These computation rules let proof developers go back and forth between different representations of the same structure or object and reuse components over them. Similarly, a relationship can be established between the refined version of a datatype to its respective unrefined one to support reusability. Ornaments, proposed by McBride in [46], can be used as a notion on a data type to define that relationship. These ornaments allow derivation of fancy data types, both indexed or annotated with additional data, from the plain ones. For example, vectors can be expressed as ornamented lists just as well lists can be expressed as ornamented numbers. Functions and proofs can be ported from one type to another if they are related through ornaments [47]. Ornaments are supported in Agda as deep embedding [48] and used in a proof reuse tool, DEVOID in coq [11]. However, it takes linear time to derive conversions between types through an ornamental relationship where Diehl *et. al.* [49] provides a constant time conversion between indexed and non-indexed types and functions over them in both directions. They use an extrinsic type theory, CDLE, to achieve this zero performance penalty, although their work is restricted to only index refinement aspect in contrast to the ornament, which also supports annotation. The same idea of proof reuse can also be applied with Voevodsky's *univalence axiom* in Homotopy Type Theory(HoTT) which provides the correct notion of *equality* to be used in theorem prover. *Univalence axiom* states type identity in HoTT to be equivalent to type equivalence [50]. Cubical Type Theory, by Cohen *et. al.* [51] provides constructive justification for this *univalence axiom*. Therefore, extensionality principles like *functional extensionality* and *propositional extensionality* can be directly proved in theory. Cubical Type Theory was further extended by Coquand *et. al.* [52] for higher inductive types which made it possible to integrate it in Agda as Cubical Agda. Nuprl has also been extened to a proof assistant, RedRpl based on cubical computational type theory [53]. Coq also followed the same path with their CoqHoTT project, which stands for Coq for HoTT [54]. Tabareau *et. al.* [55] have used HoTT, though without univalence, to provide a framework for theorem and definition transfer between equivalent types in Coq. In a nutshell, it can be said that researchers are finding new ways for entities to be equal that can be exploited to promote reuse, and recent developments in HoTT have the potential to make a breakthrough in this area of research [16].

4.3 Methods for Automation and Portability

There are several other methods and technologies that can substantially improve the efficiency of the developers beyond those that fall in the above categories. Among them, automation and portability among ITP systems are worth surveying in the scope of this work.

4.3.1 Automation

Automation combined with human-guided theorem proving can amplify human's ability to a great extent. In the earliest era of machine-assisted verification, people were mostly interested in truly automated theorem proving(ATP). However, realizing its limitation and at the same time, the power of human interaction has shifted the direction away from the automation to the more interactive arrangement. Harisson *et. al.* [56] has an interesting historical depiction on this. Nowadays, automation plays roles of moving burden away from the programmer by taking charge of repetitive, trivial tasks and reutilizing large base of previous works. Two useful and well-developed automation techniques called Theory exploration and Hammers are briefly discussed here to illustrate how they can improve the efficiency of interactive theorem proving.

Theory exploration Theory exploration is a powerful technique from the automated theorem proving (ATP) era for automatic discovery of lemmas during a proof development. There is no reason not to use the proven techniques from long research in ATP within ITP. One such combination has been used by Dramnesc *et. al.* in their work of construction of a theory of binary trees [57]. They perform their systematic exploration of theories in an automated theory exploration tool called *Theorema* [58] but prove properties for the synthesized algorithm in interactive theorem proving tool Coq. Hipster [59] is another successful initiative in this area which integrates theory exploration with Isabelle/HOL. It has two modes of operations - *explotary mode* that generates lemmas from a set data types and functions and *proof mode* that tries to discover missing lemmas during a proof creation for the current goal to be proved.

Hammers Hammers is the technique that truly exploits ATP in interactive arrangements. The target is to automatically build a proof to discharge a goal in a proof assistant. Typically, the process is decomposed into three components. Premise selector explores large formal libraries associated with proof assistants to find premises related to the conjecture. Methods like Naive Bayes and K-nearest neighbors can be used in this step for mining the libraries [60]. This component independently can be functional for users to find useful lemmas, especially when libraries are huge like Mizar Mathematical Library and trained machine learning tools like Mizar Proof Advisor [61]. The second component, Translation module, converts these selected premises and the current goal from proof assistant's logic to target ATP's logic, thus creating an ATP problem for target ATP to solve. If a proof is found by the target ATP, the final component, proof reconstructor, reduces the ATP proof to inferences of proof assistant's kernel to make the proof certifiable by the proof assistant. Several Hammers tools for different ITPS are already available including CoqHammer [62] for Coq, Sledgehammer [63] for Isabelle/HOL and HOL^{Y}Hammer [64] that is compatible with both HOL Light and HOL4 and, comes with an online extension as well [65].

4.3.2 Portability among ITP Systems

Having a mechanism to import and export proofs between different ITP systems i.e., portability of proofs, has multiple benefits in terms of increasing development efficiency. So far, we have discussed several dominant proof assistants that have been built as stand-alone programs with fundamental differences among them. Although distinct and unique features from different systems are advantageous for supporting a variety of work and personal preferences, they effectively fragmented the work base. Large libraries and huge-scale verification work from one ITP system community are now merely useful to others, and reproducing them requires a substantial amount of time. Therefore, transferring work from one domain ITP to another through portability can collectively increase the efficiency of proof engineers and facilitate collaboration and idea transfer among them.

However, proof translation from one system to another is not straightforward due to their difference in underlying theoretical foundations. The target system requires having at least richer logic than the source system so that it can accommodate all constructs in proofs written in the source. Moreover, translation of proof's statements from system to another is not the only task at hand here, though that can be classified as one form of translation. For complete portability, users of the target system should be able to recheck translated proofs at their own kernel instead of relying on the correctness of translator and source kernel. Numerous proof translators between different ITP systems can be found in literature. For example, Naumov *et. al.* [66] describes a proof translator from HOL to classical extension of Nuprl. This work is based on a series of earlier works on HOL to Nuprl translation [67, 68, 69] that provides a good insight on how theoretical foundation is built brick by brick before getting to the final translator. One step in this trail of background works was to represent HOL as a submodel of the richer Nulprl model [67] to meet the specification of having a richer target system. However, McLaughlin's work in [70] shows that having at least richer target system is not an absolute requirement for translation. In this paper, authors successfully have translated a part of Isabelle/HOL (source with richer logic) standard library into HOL light (target) with a clever workaround. The part of Isabelle/HOL logic that cannot be represented directly in HOL Light object logic is instead elaborated in HOL Light's metalanguage, OCaml using functors. Other notable works in proof translations are hol90 into CoQ [71], HOL4 and HOL Light into Isabelle/HOL [72, 73], HOL Light into Coq [74]. There is also a framework, called OpenTheory, that can transport proofs between HOL family [75]. A recent study on the interoperability of proof assistants as well as the integration of their libraries by translating them into a universal format, highlights the challenges in this field more broadly [76, 77] as well as takes the famous QED dream, a computer-based database of all mathematical knowledge formalized and machine-checked [78], one step ahead.

5 Techniques for Proof Maintenance Over Time

Proof changes over time. Not all changes can be predicted to address them through design during development, and some changes, like library updates, are out of proof engineer's control. Maintenance of large-scale proofs, thereby demands automated refactoring and repair techniques to incorporate these changes efficiently. This area of research is still in the initial phase and has lots of open problems and challenges [79]. Ideas from program refactoring and repair can be adapted into proof engineering and exploring that domain can give a head start to new researchers. However, a detail exploration of unique needs from proof engineering's side is needed as well. Here, a few tools that address the above aspects are described. It is worth noting that several of them are still in the development phase, thus have potential to offer more in future.

5.1 Proof Refactoring

Refactoring, a term coined by Opdyke in his thesis, refers to the semantic preserving transformation for the improvement of software in terms of readability, maintainability, and efficiency [80] and can be used in the domain of proof engineering as well. Proof refactoring tools can either refactor proof scripts or proof terms. Levity [81] is one of the quite mature proof refactoring tools that were developed to maintain sel4 OS kernel verification project [24]. This tool moves general lemmas to an optimal position that maximizes its reuse by specialized theorems, and it is based on the idea that lemmas should float upward through theory dependency graph. Using this tool saved them from tens of minutes to even hours worth of required work after moving a single lemma to a new theory file. POLAR [82] is a prototype generic proof script refactoring framework. It supports ten types of default refactoring and can be extended with custom ones. The high-level idea here is to transform theory to graph representation (abstract syntax) to find the appropriate part to refactor and then regain refactored theory from the transformed graph. Their framework is built on the GrGen graph rewriting engine, and it is also generic in terms of proof languages. Another tool, named Tactician [83], refactors tactic proofs in HOL light. It can fold sequences of tactics into tacticals and unfold tactical into interactive steps that can be utilized for debugging. However, this work is applicable to a small subset of ML family and does not provide a general solution for all tactic based proofs [84].

On the other hand, both refactoring tools, Chick [79] and RefactorAgda [85], operate on proof terms. The former can refactor terms in dependently typed languages like Gallina and the later refactors Agda terms. Chick works with examples, takes the original and partially refactored program and then, uses the difference between them to calculate how the whole program needs to be refactored. They also propagate these calculated changes through the rest of the program. Chick can handle insertion, modification, permutation, and deletion of subterms. Whereas RefactorAgda works on only a small subset of Agda, Baby-Agda but supports many changes including reindentation of files, renaming, function extraction, adding or removing constructor form data types, expression changing refactorings like conversion between explicit and implicit parameters, reordering of function or constructor arguments, and extracting arguments. A full list is provided in the respective paper [85]. However, all of these changes are not semantic preserving in contrast to the definition of refactoring and they can instead be classified as repair tools in that sense.

The Coq IDE, CoqPIE [86] provides some proof script refactoring support from the IDE level such as lemma extraction and replay. In replay, proof scripts before and after modification are parsed into abstract syntax trees and can be updated to incorporate fix like correcting hypothesis references. Lemma extraction can transform subgoals into separate lemmas. More supports can be provided from the IDE level, thus has much room for work in this domain.

5.2 Proof Repair

As discussed in previous sections, proofs can break due to any change in the definition or in the theorem specification they depend on. Thus one of the major goals of proof design is to make proof resilient in the face of these changes. However, one can predict so much in advance, thus requires repair tools to adapt proofs to unpredictable changes. A Proof repair tool ideally should be able to automatically create a reusable patch when any change to the specification or definition happens and apply them to fix dependent proofs. There is a Coq plugin, PUMPKIN PATCH [87] that can do part of this process. It takes examples patches of how to adapt dependent proofs in response to some change in definition or theorem and then generalizes these example patches into a reusable one. It still cannot apply the patch automatically and depends on example patches. Recently they partnered with Galois, Inc. to add more functionalities to their tool.

6 Tooling and IDE Support

Tooling and IDE support from proof assistants aid in proof development by providing means for automation, theorem searching, refactoring, and repairing within the system as well as by helping in project management through integrated development environments. Several of these tooling supports have already been discussed in previous sections. In section 4, automated theorem proving tool Hammer, library mining tool Mizar Proof Advisor, proof reuse tool DEVOID have been covered. Section 5 includes a description of proof refactoring tools Levity, POLAR, Tactician, Chick, RefactorAgda, and proof repairing tool PUMPKIN PATCH. However, some tools also by default support data types and theorem search. For example, Coq has *Search* command that can find relevant proofs form the standard library. Isabelle provides similar support through *find_consts* and *find_theorems* commands. These tooling supports can speed up the development process and improve maintainability to a great extent (see section 4 and 5 for detail). Some of these tools are not matured yet, though, and still have room for improvement.

Proof assistants now alos have IDEs for development, which is a significant improvement over the initial REPL(Read Eval Print Loop) interface they had. CoqIDE and Isabelle/jEdit are the most popular IDEs for Coq and Isabelle. Proof general is another popular IDE within the Coq community, although it can be used with other proof assistants like Lego, Isabelle. It can also be extended with new functionalities like auto-completion. CoqIDE, along with another lesser-known IDE, Coqoon for Coq have support for project management. IDEs are also increasingly promoting asynchronous development. Previously, it was not possible to have a checker verifying some proofs when the user is modifying others. Now, it is supported in CoqIDE, Coqoon, as well as in Isabelle/jEdit. There is an interesting fact about Nuprl, however. While all most ITP systems followed the path from REPL through Emac to IDEs, Nuprl was launched with a graphical interface. Agda, however, does not have any IDE support. Proof editing in Agda is done through Emacs or Atom. In Emacs mode, as described in section 2, the user can define holes in proof and have it type-checked and then fill in those holes later.

Both tooling and IDE has come a long way for proof assistants and can be improved to better support large-scale development. There is a possibility that more productivity tools like refactoring and repairing would emerge in the coming years [16]. Another support that is scarce at this moment is debugging tools for proof assistants. Notably, the Tactic-based proofs do not allow any user interaction while running tactics or tacticals. If a tactic or tactical fails, the proof state goes back to the step before running them without providing any information on how it failed. It could be helpful to know the intermediate information for debugging purposes as well as for deciding the next move. There are some works in the tactic generalization that have the same idea and can be adapted for tooling as well [44]. Large scale development also benefits from advanced IDEs. Tools like mentioned above can be integrated into IDE as plugins. The idea for useful features in development can be borrowed from software engineering and implemented in extensible IDEs like Proof General. Moreover, thorough usability analyses for proof development need to be performed to identify domain-specific requirements.

7 Conclusion

This survey explores research on practical aspects of ITP systems to accumulate identified domain-specific challenges in large scale proof development as well as organizes proposed methodologies to tackle these challenges based on the proof engineering concerns they address. Efficiency and maintainability are two main challenges identified by researchers and proof engineers involved in past large-scale verification projects. Proofs in scaled development require to be adaptable i.e., robust and easily modifiable, well-organized, and reusable to promote efficient and maintainable proof development. Automation and portability of proofs among different ITP systems improve the performance of proof developers, thereby the efficiency of overall development. Proofs also need maintenance in the face of inevitable changes over time and incorporating these changes in large-scale development is another challenging task. Refactoring and repair are two automated techniques that researchers are actively working on to adapt or recover proofs automatically when change happens. Proof assistants are now being integrated with these techniques to provide better tooling support within the system. Many present ITP systems have Integrated Development Environments(IDE) too. Some of them provide project management systems and asynchronous development capability and all are evolving to better support the development process.

Active research are now being carried out on most of these aspects mentioned above and they have lots of room for improvement. Proof reusability through type equivalence as well as proof refactoring and repair have the most promising prospect for future research work. Well-developed methods from software engineering can also be borrowed to advance proof engineering. However, proof development is inherently different from software development. Hence, comprehensive usability studies need to be performed to draw out specific requirements for this domain.

References

[1] X. Leroy, "Formal certification of a compiler back-end or: Programming a compiler with a proof assistant," *SIGPLAN Not.*, vol. 41, no. 1, p. 42–54, Jan. 2006. [Online]. Available: https://doi.org/10.1145/1111320.1111042

[2] N. G. de Bruijn, "Automath, a language for mathematics," in *Automation of Reasoning*. Springer, 1983, pp. 159–200.

[3] P. Martin-Löf and G. Sambin, *Intuitionistic type theory*. Bibliopolis Naples, 1984, vol. 9.

[4] T. Coquand and G. Huet, "The calculus of constructions," *Information and Computation*, vol. 76, no. 2, pp. 95 – 120, 1988. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0890540188900053

[5] T. C. D. Team, "The coq proof assistant, version 8.10.0," Oct. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.3476303

[6] D. S. Scott, "A type-theoretical alternative to iswim, cuch, owhy," *Theoretical Computer Science*, vol. 121, no. 1-2, pp. 411–440, 1993.

[7] H. Geuvers, "Proof assistants: History, ideas and future," *Sadhana*, vol. 34, no. 1, pp. 3–25, 2009.

[8] M. Wenzel, L. C. Paulson, and T. Nipkow, "The isabelle framework," in *International Conference on Theorem Proving in Higher Order Logics*. Springer, 2008, pp. 33–38.

[9] M. J. C. Gordon and T. F. Melham, *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. USA: Cambridge University Press, 1993.

[10] F. Wiedijk, "Formal proof sketches," in *International Workshop on Types for Proofs and Programs*. Springer, 2003, pp. 378–393.

[11] T. Ringer, N. Yazdani, J. Leo, and D. Grossman, "Ornaments for Proof Reuse in Coq," in *10th International Conference on Interactive Theorem Proving (ITP 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), J. Harrison, J. O'Leary, and A. Tolmach, Eds., vol. 141. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, pp. 26:1–26:19. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2019/11081

[12] F. Wiedijk, "Comparing mathematical provers," in *International Conference on Mathematical Knowledge Management*. Springer, 2003, pp. 188–202.

[13] J. Harrison, J. Urban, and F. Wiedijk, "History of interactive theorem proving." in *Computational Logic*, vol. 9, 2014, pp. 135–214.

[14] H. Barendregt and H. Geuvers, "Proof-assistants using dependent type systems." *Handbook of automated reasoning*, vol. 2, pp. 1149–1238, 2001.

[15] A. Abel, A. Vezzosi, and T. Winterhalter, "Normalization by evaluation for sized dependent types," *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, Aug. 2017. [Online]. Available: https://doi.org/10.1145/3110277

[16] T. Ringer, K. Palmskog, I. Sergey, M. Gligoric, and Z. Tatlock, "Qed at large: A survey of engineering of formally verified software," *Foundations and Trends® in Programming Languages*, vol. 5, no. 2-3, pp. 102–281, 2019. [Online]. Available: http://dx.doi.org/10.1561/2500000045

[17] A. Yushkovskiy and S. Tripakis, "Comparison of two theorem provers: Isabelle/hol and coq," *CoRR*, vol. abs/1808.09701, 2018. [Online]. Available: http://arxiv.org/abs/1808.09701

[18] A. Chlipala, *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. [Online]. Available: http://mitpress.mit.edu/books/certified-programming-dependent-types

[19] U. Norell, "Dependently typed programming in agda," in *Koopman P*, R. Plasmeijer and D. Swierstra, Eds. vol 5832. Springer, Berlin, Heidelberg: Advanced Functional Programming. AFP 2008. Lecture Notes in Computer Science, 2009.

[20] F. Haftmann and T. Nipkow, "Code generation via higher-order rewrite systems," in *In Functional and Logic Programming, 10th International Symposium: FLOPS 2010, volume 6009 of Lecture Notes in Computer Science*. Springer, 2010.

[21] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, p. 107–115, Jul. 2009. [Online]. Available: https://doi.org/10.1145/1538788.1538814

[22] A. SIGPLAN, "Most influential popl paper award." [Online]. Available: http://www.sigplan.org/Awards/POPL

[23] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, and et al., "Sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 207–220. [Online]. Available: https://doi.org/10.1145/1629575.1629596

[24] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an os microkernel," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, 02 2014.

[25] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. USA: USENIX Association, 2014, p. 305–320.

[26] D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. Anderson, "Planning for change in a formal verification of the raft consensus protocol," in *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, ser. CPP 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 154–165. [Online]. Available: https://doi.org/10.1145/2854065.2854081

[27] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo, "Certikos: An extensible architecture for building certified concurrent os kernels," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. USA: USENIX Association, 2016, p. 653–669.

[28] R. Kaivola and K. Kohatsu, "Proof engineering in the large: formal verification of pentium?4 floating-point divider," *International Journal on Software Tools for Technology Transfer*, vol. 4, no. 3, pp. 323–334, 2003. [Online]. Available: https://doi.org/10.1007/s10009-002-0081-6

[29] B. Delaware, W. Cook, and D. Batory, "Product lines of theorems," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 595–608. [Online]. Available: https://doi.org/10.1145/2048066.2048113

[30] P. Curzon, "The importance of proof maintenance and reengineering," in *Int. Workshop on Higher Order Logic Theorem Proving and Its Applications: B-Track*, 1995, pp. 17–32.

[31] A. Chlipala, "Modular development of certified program verifiers with a proof assistant,," *J. Funct. Program.*, vol. 18, no. 5-6, pp. 599–647, 2008. [Online]. Available: https://doi.org/10.1017/S0956796808006904

[32] J.-C. Filliâtre and P. Letouzey, "Functors for Proofs and Programs," in *13th European Symposium on Programming, ESOP 2004*, ser. Lecture Notes in Computer Science, D. Schmidt, Ed. Barcelona, Spain: Springer, Feb. 2004, pp. 370–384. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00150913

[33] A. W. Appel, "Efficient verified red-black trees," 2011. [Online]. Available: https://www.cs.princeton.edu/~appel/papers/redblack.pdf

[34] C. Ballarin, "Locales and locale expressions in isabelle/isar," in *Berardi S*, M. Coppo and F. Damiani, Eds. vol 3085. Springer, Berlin, Heidelberg: Types for Proofs and Programs. TYPES 2003. Lecture Notes in Computer Science, 2004.

[35] F. Kamm"uller, M. Wenzel, and L. C. Paulson, “Locales a sectioning concept for isabelle,” in *Bertot Y*, G. Dowek, L. Théry, A. Hirschowitz, and C. Paulin, Eds. vol 1690. Springer, Berlin, Heidelberg: Theorem Proving in Higher Order Logics. TPHOLs 1999. Lecture Notes in Computer Science, 1999.

[36] C. Ballarin, “Tutorial to locales and locale interpretation,” *Contribuciones científicas en honor de Mirian Andrés Gómez, 2010-01-01, ISBN 978-84-96487-50-5, pags. 123-140*, 01 2010.

[37] F. Rabe and C. Schürmann, “A practical module system for lf,” in *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, ser. LFMTP ’09. New York, NY, USA: Association for Computing Machinery, 2009, p. 40–48. [Online]. Available: https://doi.org/10.1145/1577824.1577831

[38] S. D. Team, “Structtact.” [Online]. Available: http://github.com/uwplse/StructTact

[39] O. Pons, “Conception et réalisation d’outils d’aide au développement de grosses théories dans les systèmes de preuves interactifs,” Ph.D. dissertation.

[40] O. Boite, “Proof Reuse with Extended Inductive Types,” in *TPHOLs 2004*, ser. LNCS, vol. 3223, X, France, Jan. 2004, pp. 50–65. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01124944

[41] A. Mulhern, ““proof weaving”,” in *In Proceedings of the First Informal ACM SIGPLAN Workshop on Mechanizing Metatheory*, 2006.

[42] S. Berghofer and T. Nipkow, “Proof terms for simply typed higher order logic,” in *Theorem Proving in Higher Order Logics*, M. Aagaard and J. Harrison, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 38–52.

[43] J. E. Caplan and M. T. Harandi, “A logical framework for software proof reuse,” *SIGSOFT Softw. Eng. Notes*, vol. 20, no. SI, p. 106–113, Aug. 1995. [Online]. Available: https://doi.org/10.1145/223427.211821

[44] A. Felty and D. J. Howe, “Generalization and reuse of tactic proofs,” in *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, ser. LPAR ’94. Berlin, Heidelberg: Springer-Verlag, 1994, p. 1–15.

[45] G. Barthe and O. Pons, “Type isomorphisms and proof reuse in dependent type theory,” in *International Conference on Foundations of Software Science and Computation Structures*. Springer, 2001, pp. 57–71.

[46] C. McBride, ““ornamental algebras,” in *algebraic ornaments”*, url, 2011. [Online]. Available: http://plv.mpi-sws.org/plerg/papers/mcbride-ornaments-2up.pd

[47] P.-E. Dagand and C. McBride, “Transporting functions across ornaments,” *SIGPLAN Not.*, vol. 47, no. 9, p. 103–114, Sep. 2012. [Online]. Available: https://doi.org/10.1145/2398856.2364544

[48] T. Williams, P.-E. Dagand, and D. Rémy, “Ornaments in practice,” in *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming*, ser. WGP ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 15–24. [Online]. Available: https://doi.org/10.1145/2633628.2633631

[49] L. Diehl, D. Firsov, and A. Stump, “Generic zero-cost reuse for dependent types,” *Proc. ACM Program. Lang.*, vol. 2, no. ICFP, Jul. 2018. [Online]. Available: https://doi.org/10.1145/3236799

[50] S. Awodey, Á. Pelayo, and M. A. Warren, “Voevodsky’s univalence axiom in homotopy type theory,” *Notices of the AMS*, vol. 60, no. 9, pp. 1164–1167.

[51] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg, "Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom," in *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), T. Uustalu, Ed., vol. 69. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 5:1–5:34. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2018/8475

[52] T. Coquand, S. Huber, and A. Mörtberg, "On higher inductive types in cubical type theory," *CoRR*, vol. abs/1802.01170, 2018. [Online]. Available: http://arxiv.org/abs/1802.01170

[53] C. Angiuli, E. Cavallo, K.-B. Hou, R. Harper, and J. Sterling, "The redprl proof assistant (invited paper)," 07 2018.

[54] C. Project., "The coqhott project," 2015-2020. [Online]. Available: http://coqhott.gforge.inria.fr

[55] N. Tabareau, E. Tanter, and M. Sozeau, "Equivalences for free: Univalent parametricity for effective transport," *Proc. ACM Program. Lang.*, vol. 2, no. ICFP, Jul. 2018. [Online]. Available: https://doi.org/10.1145/3236787

[56] J. Harrison, J. Urban, and F. Wiedijk, "History of interactive theorem proving," in *Computational Logic*, ser. Handbook of the History of Logic, J. H. Siekmann, Ed. North-Holland, 2014, vol. 9, pp. 135 – 214. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780444516244500046

[57] I. Drâmnesc, T. Jebelean, and S. Stratulat, "Theory exploration of binary trees," in *2015 IEEE 13th International Symposium on Intelligent Systems and Informatics (SISY)*, Sep. 2015, pp. 139–144.

[58] B. Buchberger, A. Crăciun, T. Jebelean, L. Kovács, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger, "Theorema: Towards computer-aided mathematical theory exploration," *Journal of Applied Logic*, vol. 4, no. 4, pp. 470 – 504, 2006, towards Computer Aided Mathematics. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1570868305000716

[59] M. Johansson, D. Rosén, N. Smallbone, and K. Claessen, "Hipster: Integrating theory exploration in a proof assistant," *CoRR*, vol. abs/1405.3426, 2014. [Online]. Available: http://arxiv.org/abs/1405.3426

[60] J. Blanchette, C. Kaliszyk, L. Paulson, and J. Urban, "Hammering towards qed," *Journal of Formalized Reasoning*, vol. 9, no. 1, pp. 101–148, 2016. [Online]. Available: https://jfr.unibo.it/article/view/4593

[61] J. Urban, "Mptp–motivation, implementation, first experiments," *Journal of Automated Reasoning*, vol. 33, no. 3-4, pp. 319–339, 2004.

[62] L. Czajka and C. Kaliszyk, "Hammer for coq: Automation for dependent type theory," *Journal of Automated Reasoning*, vol. 61, no. 1, pp. 423–453, 2018. [Online]. Available: https://doi.org/10.1007/s10817-018-9458-4

[63] L. Paulson and J. Blanchette, "Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers," 02 2015.

[64] C. Kaliszyk and J. Urban, "Learning-assisted automated reasoning with flyspeck," *Journal of Automated Reasoning*, vol. 53, no. 2, pp. 173–213, 2014. [Online]. Available: https://doi.org/10.1007/s10817-014-9303-3

[65] ——, “Hol(y)hammer: Online atp service for hol light,” *Mathematics in Computer Science*, vol. 9, no. 1, pp. 5–22, 2015. [Online]. Available: https://doi.org/10.1007/s11786-014-0182-0

[66] P. Naumov, S. MO., and J. Meseguer, “The hol/nuprl proof translator,” in *Boulton R*, J. P. B. J., Ed. vol 2152. Springer, Berlin, Heidelberg: Theorem Proving in Higher Order Logics. TPHOLs 2001. Lecture Notes in Computer Science, 2001.

[67] D. J. Howe, “Semantic foundations for embedding hol in nuprl,” in *Wirsing M*, M. Nivat, Ed. vol 1101. Springer, Berlin, Heidelberg: Algebraic Methodology and Software Technology. AMAST 1996. Lecture Notes in Computer Science, 1996.

[68] ——, “Importing mathematics from hol into nuprl,” in *Goos G*, J. Hartmanis, J. van Leeuwen, J. von Wright, J. Grundy, and J. Harrison, Eds. vol 1125. Springer, Berlin, Heidelberg: Theorem Proving in Higher Order Logics. TPHOLs 1996. Lecture Notes in Computer Science, 1996.

[69] A. P. Felty and D. J. Howe, “Hybrid interactive theorem proving using nuprl and hol,” in *Automated Deduction—CADE-14*, W. McCune, Ed. vol 1249. Springer, Berlin, Heidelberg: CADE 1997. Lecture Notes in Computer Science (Lecture Notes in Artificial Intelligence), 1997.

[70] S. McLaughlin, “An interpretation of isabelle/hol in hol light,” in *Proceedings of the Third International Joint Conference on Automated Reasoning*, ser. IJCAR’06. Berlin, Heidelberg: Springer-Verlag, 2006, p. 192–204. [Online]. Available: https://doi.org/10.1007/11814771_18

[71] E. Denney, “A prototype proof translator from hol to coq,” in *Aagaard M*, J. Harrison, Ed. vol 1869. Springer, Berlin, Heidelberg: Theorem Proving in Higher Order Logics. TPHOLs 2000. Lecture Notes in Computer Science, 2000.

[72] S. Obua and S. Skalberg, “Importing hol into isabelle/hol,” in *Furbach U*, N. Shankar, Ed. vol 4130. Springer, Berlin, Heidelberg: Automated Reasoning. IJCAR 2006. Lecture Notes in Computer Science, 2006.

[73] C. Kaliszyk and A. Krauss, “Scalable lcf-style proof translation,” in *Proceedings of the 4th International Conference on Interactive Theorem Proving*, ser. ITP’13. Berlin, Heidelberg: Springer-Verlag, 2013, p. 51–66. [Online]. Available: https://doi.org/10.1007/978-3-642-39634-2_7

[74] C. Keller and B. Werner, “Importing hol light into coq,” in *International Conference on Interactive Theorem Proving*. Springer, 2010, pp. 307–322. [Online]. Available: https://doi.org/10.1007/978-3-642-14052-5_22

[75] J. Hurd, “The opentheory standard theory library,” in *Bobaru M*, K. Havelund, G. J. Holzmann, and R. Joshi, Eds. vol 6617. Springer, Berlin, Heidelberg: NASA Formal Methods. NFM 2011. Lecture Notes in Computer Science, 2011. [Online]. Available: https://doi.org/10.1007/978-3-642-20398-5_14

[76] M. Kohlhase and F. Rabe, “Qed reloaded: Towards a pluralistic formal library of mathematical knowledge,” *J. Formalized Reasoning*, vol. 9, pp. 201–234, 2016.

[77] ——, “Experiences from exporting major proof assistant libraries,” 2020. [Online]. Available: https://kwarc.info/people/frabe/Research/KR_oafexp_20.pdf

[78] “The qed manifesto,” in *Proceedings of the 12th International Conference on Automated Deduction*, ser. CADE-12. Berlin, Heidelberg: Springer-Verlag, 1994, p. 238–251.

[79] V. Robert, “Front-end tooling for building and maintaining dependently-typed functional programs,” Ph.D. dissertation, UC San Diego, 2018.

[80] W. F. Opdyke, “Refactoring object-oriented frameworks,” Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[81] T. Bourke, M. Daum, G. Klein, and R. Kolanski, “Challenges and experiences in managing large-scale proofs,” in *Intelligent Computer Mathematics*, J. Jeuring and others Jeuring J. al, Eds. vol 7362. Springer, Berlin, Heidelberg: CICM 2012. Lecture Notes in Computer Science, 2012.

[82] D. Dietrich, I. Whiteside, and D. Aspinall, “Polar: A framework for proof refactoring,” in *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, ser. Lecture Notes in Computer Science, K. L. McMillan, A. Middeldorp, and A. Voronkov, Eds., vol. 8312. Springer, 2013, pp. 776–791. [Online]. Available: https://doi.org/10.1007/978-3-642-45221-5_52

[83] M. Adams, “Refactoring proofs with tactician,” in *Revised Selected Papers of the SEFM 2015 Collocated Workshops on Software Engineering and Formal Methods - Volume 9509*. Berlin, Heidelberg: Springer-Verlag, 2015, p. 53–67. [Online]. Available: https://doi.org/10.1007/978-3-662-49224-6_6

[84] Y. Lin, G. Grov, and R. Arthan, “Understanding and maintaining tactics graphically OR how we learned that a diagram can be worth more than 10k loc,” *CoRR*, vol. abs/1610.05593, 2016. [Online]. Available: http://arxiv.org/abs/1610.05593

[85] K. Wibergh, “Automatic refactoring for agda,” 2018.

[86] K. Roe and S. F. Smith, “Coqpie: An IDE aimed at improving proof development productivity - (rough diamond),” in *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, ser. Lecture Notes in Computer Science, J. C. Blanchette and S. Merz, Eds., vol. 9807. Springer, 2016, pp. 491–499. [Online]. Available: https://doi.org/10.1007/978-3-319-43144-4_32

[87] T. Ringer, N. Yazdani, J. Leo, and D. Grossman, “Adapting proof automation to adapt proofs,” in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 115–129. [Online]. Available: https://doi.org/10.1145/3167094