Noname manuscript No.
(will be inserted by the editor)

Experiences from Exporting Major Proof Assistant Libraries

Michael Kohlhase (corresponding author) · Florian Rabe

Received: 2020 / Accepted: 2021

Abstract The interoperability of proof assistants and the integration of their libraries is a highly valued but elusive goal in the field of theorem proving. As a preparatory step, in previous work, we translated the libraries of multiple proof assistants, specifically the ones of Coq, HOL Light, IMPS, Isabelle, Mizar, and PVS into a universal format: OMDoc/MMT.

Each translation presented great theoretical, technical, and social challenges, some universal and some system-specific, some solvable and some still open. In this paper, we survey these challenges and compare and evaluate the solutions we chose.

We believe similar library translations will be an essential part of any future system interoperability solution and our experiences will prove valuable to others undertaking such efforts.

1 Introduction

Motivation The QED manifesto [2] of 1994 urged the automated reasoning community to work towards a universal, computer-based database of all mathematical knowledge, strictly formalized in logic and supported by proofs that can be checked mechanically. The QED database was intended as a communal resource that would guide research and allow the evaluation of automated reasoning tools and systems. This database was never realized, but the interoperability of proof assistants and the integration of their libraries has remained a highly valued but elusive goal.

This is despite the large and growing need for more integrated and easily reusable libraries of formalizations. For example, the Flyspeck project [20] built a formal proof of the Kepler conjecture in HOL Light. But it relies on results achieved using Isabelle's reflection mechanism, which cannot be easily recreated in HOL Light. And that is only an integration problem between two systems using the same foundation — users of theorem provers often approach (and occasionally exasperate) developers with requests to be able to use, e.g., Coq's tactics together

Computer Science, FAU Erlangen-Nürnberg
E-mail: michael.kohlhase@fau.de · E-mail: florian.rabe@fau.de

with Isabelle's sledgehammer tool, requests that sound simple to users but are known to be infeasible for developers, who understand the underlying principles.

Problem and Related Work No strong tool support is available for any such integration. In fact, merging libraries can hardly even be attempted because we lack satisfactory mechanisms to translate across languages or into a common language. Even worse, most integration attempts currently falter already when trying to *access* the libraries in the first place. The libraries consist of text files in languages optimized for fast and convenient writing by human users. Consequently, highly non-trivial algorithms for parsing, type reconstruction, and theorem proving have been developed to build the corresponding abstract data structures. This has the effect that for each library, there is essentially only a single system able to read it. Moreover, these systems are typically realized as read-evaluate-print interfaces to the foundation, where any user interaction causes a state transition in the kernel, whose results must be interpretable by the users. Therefore, any integration requires the theorem prover to support *exporting* libraries in formats that can be read by external tools such as other proof assistants, interoperability middleware, or knowledge management services.

But even when exports exist, there are two major problems. Firstly, most exports contain the elaborated low-level data structures that are suitable for the kernel and not the high-level structure that is seen by the user. The latter usually corresponds more closely to the informal domain knowledge and is therefore more valuable for reuse. Secondly, the export code quickly becomes out-of-date as new features are added to the main system. The only exception are exports that are actively maintained by the developers of the respective theorem prover, but this is rarely the case.

Therefore, there are only a few examples of successful transports of a library between two proof assistants. Some have been realized as *ad-hoc logic translations*, typically in special situations. [33] translates from HOL Light [22] to Coq [12] and [50] to Isabelle/HOL. Both translations benefit from the well-developed HOL Light export and the simplicity of the HOL Light foundation. [39] translates from Isabelle/HOL [48] to Isabelle/ZF [54]. Here import and export were aided by the use of the same logical framework. The Coq library has been imported into Matita, aided by the fact that both use very similar foundations. The recent Lean system includes APIs that make translations into other formats relatively easy; this was mostly used for independent proof checking and an integration with Mathematica [41]. In [57], the Nuprl language was represented in Coq for the purpose of verifying Nuprl proofs. The OpenTheory format [24] was developed to facilitate sharing between HOL-based systems but has not been used extensively.

Alternatively, one can use *logical framework-based transports*, where the target system is a logical framework. This approach was used by us in the work we present here. It is also used by the Dedukti group, e.g., for HOL Light in [7] and for Coq in [6]. In principle, the logical framework can serve as a uniform intermediate data structure via which libraries can be moved into other proof assistants. Such translations were built in [60] from a representation of HOL in LF to one of Nurpl and similarly in [10] for a large set of logics. But these approaches lived only in the logical framework and lacked a connection to the actual systems and their libraries. Recently small arithmetic libraries were transported across actual systems using Dedukti as an intermediate system [61].

Of course, it is also imaginable that the evolution of proof assistants will lead to a future with only $2-3$ major systems, whose growing developer and user communities have the resources to maintain semi-automated migration workflows for each other's libraries. However, even in that scenario the costs are high, and a systematic investigation of the associated difficulties and pitfalls is helpful.

Contribution In [37], we proposed a major project of extending representations of proof assistant logics in a logical framework and exporting their *entire* libraries into a universal format. While far from a final QED interoperability solution, it constituted a critical step towards building interoperability and knowledge management applications.

Since then, this proposal was put into practice in the context of the OAF project (Open Archive of Formalizations), including exports for some of the biggest existing libraries. To use resources efficiently, we chose representative proof assistants: one each based on higher-order logic (namely HOL Light [22]), constructive type theory (Coq [12]), an undecidable type theory (PVS [51]), and set theory (Mizar [62]), as well as one based on a logical framework (Isabelle [53]) and one based on axiomatic specification (IMPS [16]). We discuss some notable omissions in Section 9.2. All six exports were presented individually before: HOL Light in [31], Mizar in [27], PVS in [36], IMPS in [8], Coq in [46], and Isabelle in [38]. For simplicity, we will refer to these as "our exports" in the sequel even though each one was developed with different collaborators.

In the process of these five years of work, we have accumulated a lot of knowledge that is complementary to the individual papers, and in this article we report on our experiences with this enterprise. Circumstances led us to try several very different approaches in all these exports. While the previous papers presented the logical details of each approach in depth, the present paper abstracts from the technicalities, discusses the general challenges, and compares the approaches. It describes in general the problems (solved or remaining), possible solutions, and future priorities concerning these exports. It does not subsume the previous papers, and we repeat citations already present in the previous papers only if they are of particular interest here. Instead, we focus on the details that are relevant for comparing and evaluating the approaches and try to record and pass on the knowledge/lessons that will help other researchers attempting future proof assistant exports. Thus, one can see the individual exports as experiments that created data, and the current article as the one that interprets and draws conclusions from that data.

While library integration is the ultimate goal of our work, the present paper focuses only on the technical aspects of exporting from a prover into some other representation. We defer the problem of actually making a semantic between multiple such exports to future work.

Overview Section 2 reports on the common aspects of exporting theorem prover libraries, and Sections 3 to 8 discuss system-specific challenges, solutions, and related work. Section 9 concludes the paper and points out future work. The entire text of this paper is new with the exception that the sections on the individual systems also include short high-level summaries of the respective export to make the paper self-contained.

2 General Considerations

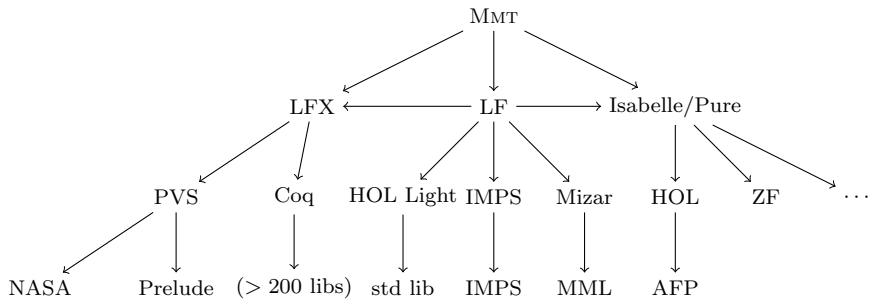

Fig. 1 Major Libraries in MMT as a Universal Framework

We discuss multiple aspects of exports. Eventually, Table 1 gives an overview of how these aspects are realized in our six exports.

2.1 General Approach

We use OMDoc [35] as the standard XML format for holding the library exports. Its semantics and API are provided by the MMT system [56,55]. MMT provides implementations of logical frameworks such as LF [21] and extensions thereof (collectively named LFX; see [43]) that we have developed for representing theorem prover logics adequately.

The general pattern for exporting the library $\mathcal{L}$ of a theorem prover $\mathcal{T}$ is as follows:

1. Within MMT, we choose a logical framework in which we can define the logic underlying $\mathcal{T}$ as a theory $\mathcal{T}^*$. Preferably, this is LF but in practice often a stronger, sometimes custom-designed framework is needed.
2. We produce a **system-near export** $\mathcal{L}'$ of $\mathcal{L}$, usually using an ad hoc schema for a general purpose data representation format like XML or JSON. This can be obtained by instrumenting $\mathcal{T}$ or by traversing the internal data structures after processing the library.
3. We read $\mathcal{L}'$ into MMT and
   - standardize or mimic its statement syntax in MMT,
   - represent its expression syntax relative to $\mathcal{T}^*$,
   - serialize the result as OMDoc yielding $\mathcal{L}^*$.

The resulting collection of libraries is shown in Figure 1. Here the third level from the top shows the theorem prover languages and the bottom level their libraries. For example, the NASA library, the Mizar Mathematical Library, and the Archive of Formal Proofs are the major libraries of PVS, Mizar, and Isabelle. The second level shows the logical framework in which we represented the prover language: mostly LF [21] or extensions thereof (LFX); because the Isabelle system is a logical framework itself, it occurs at this level. All of these framework are realized within MMT.

Crucially, this design enables a separation of skill sets: The first step above can be best executed by an expert for the logic of $\mathcal{T}$ (and LF/LFX), whereas the second step can be best execute by an expert for $\mathcal{T}$, often a core developer. This system expert is required as no system is documented well enough to make an export feasible without an expert on the inside. The system-near export can be implemented as a stand-alone component of the $\mathcal{T}$, e.g., as a traverser for the internal data structures; or it may be a major part of the system, possibly developed over years or requiring changes all across the codebase. Incidentally, in the latter case, we were often not able to precisely quantify the effort in terms of developer time or lines of code.

The third step can be best executed by an Mmt expert using a direct line of communication to the $\mathcal{T}$ expert to discuss the meaning and details of $\mathcal{L}'$. A good practice is for the two of them to jointly write or document the schema for the system-near export. A two-week research visit proved to be the minimal investment necessary to synchronize understanding of the schema and to specify (not implement) the export. Then the implementation usually consists of a straightforward parser of the system-near export followed by a traversal to translate it to Mmt. This part is usually implemented as a self-contained add-on to Mmt of relatively small size — usually a few thousand lines of code depending on how many productions the grammar for the internal data structures has. An exception was HOL Light, where the data structures were so simple that we included the second part in the prover instrumentation.

This separation of steps allows maintaining the generation of $\mathcal{L}'$ together with the $\mathcal{T}$ release process and that of $\mathcal{L}^*$ with the Mmt release process. This has proved to be the best recipe against the bit-rot of exporters, which is one of the biggest problems in practice.

Notably, the involved tasks are badly incentivized: they are very difficult, very work-intensive, and tend to produce brittle implementations that generate only one paper. This is an open problem with academic system development in general and prover library exports in particular — our exports were only possible due to well-orchestrated collaborations that drove the creation and documentation of the system-near exports. For example, a big part of our PVS export was working with the developers to document and debug the generation of XML files. In the case of Coq, Claudio Sacerdoti Coen's kernel export plugin [59] was developed specifically for the occasion, the Isabelle export was enabled by heavy investments by Makarius Wenzel into the PIDE infrastructure — both efforts that required months of full-time work.

2.2 Compositionality and Trustworthiness

An important question is whether a library export is correct, particularly in the realistic scenario where we can translate a library to another system only without proofs (because translating proofs is generally much harder than translating everything else). Moreover, even if all proofs can be exported and rechecked (both of which are problematic, see below), there is no guarantee that they were exported correctly. In fact, typical exports go through multiple bugs where proofs are falsely reported as successfully rechecked. For example, even a bug as blatant as accidentally generating all theorem files as empty strings (which then trivially recheck)

can be surprisingly hard to notice because these exports involve large sets of large files often containing formalizations from domains that neither the prover expert nor the MMT expert understand.

Rarely used or experimental language features pose a related problem. Often the prover expert notices missing cases in her export only when the feature is actually used in a library, which may be very far down the line. For example, in our Coq export, we encountered kernel features that were not used at all in any of the libraries that we could build with the latest Coq version — an issue we only noticed because we actually looked for uses of the feature in order to reverse-engineer the intended semantics from the way it was used.

To ensure the correctness of exporters, we found the skill-separating export approach above extremely valuable. It allows for the system-near export to be relatively trustworthy as it is a straightforward serialization of the kernel data structures, and it pushes all logical transformations (e.g., eliminating subtyping) to a later phase. Mixing the serialization of the data structures with such logical transformations is not only undesirable (because information is lost) but has also proved error-prone because the former is a major implementation effort and the latter extremely difficult theoretically.

Even so, we found that any logical transformation in later phases should remain simple and compositional: any non-compositional transformation is usually a huge theoretical challenge, ends up incomplete or hacky because the theory did not cover all cases that are actually used in the prover, and is the first thing to break upon new prover releases.

Thus, we have opted to avoid non-compositional transformations — if in doubt, we mimicked the language feature in MMT by extending the logical framework. For example, we chose not to eliminate PVS subtyping and instead developed a logical framework that supports subtyping directly. Similarly, we did not eliminate Coq universe constraints or Mizar structures. Thus, any feature eliminations (which may be necessary, e.g., to translate into a less expressive language) are optional and can be delayed until needed.

2.3 Generating the Prover-Near Export

There are multiple ways to **set up the export** of libraries. The most direct approach of exporting from the source files basically never works in practice because it would essentially require reimplementing the entire prover. Instead, only exports that use the kernel data structures are feasible. There are two ways to do this: *Instrumenting* the kernel means that the library is checked by the kernel, and kernel hooks are used to generate the export. Thus, exporting requires rechecking the entire library. (If proof exports are not needed, it may be possible to tweak the kernel to skip all proofs, which is the most expensive part.) But if the kernel is small and well-structured, instrumentation is relatively easy to implement and maintain. *Traversing* the kernel data structures means that we first check a unit of source code (e.g., file) and then traverse its internal representation to generate the export. The code for this can be written as a *plugin* to or by *modifying* the sources of the system.

Alternatively, we can use already existing kernel-generated files (e.g., binary files for Coq or XML files for Mizar). This requires no modifications of the kernel

and is faster. We can distinguish two cases here. Firstly, if the kernel-generated files are already *used internally* (as for Coq and Mizar), the code generating them is *integrated* with the prover, thus maintenance is assured and bit-rot avoided. But this approach is limited to the information that the kernel generated, which is often not exactly what one is interested in for the export. And the file structure may be subject to change, thus breaking any tool that picks up on them. For that reason, we instrumented the Coq kernel even though it already generates binary files. Secondly, if the kernel-generated files are not used internally but are *only generated for export purposes* (as for PVS), it is easier to negotiate changes to their format (e.g., to include additional information). But in this case there is less assurance that the files were generated correctly in the first place. In that case, an extended debug cycle may be necessary until the files are indeed generated correctly. But that is still much easier than building an instrumentation from scratch.

Because the prover-near exports are (intentionally) highly system-specific and used only as an intermediate representation before exporting into the actual target format, they can use any easily parseable structured data format such as JSON or XML. In our experience, text files work well, and binary files work only if the prover supplies an API for reading them. In practice, the format is largely irrelevant — usually the documentation of the XML or JSON schema is the bottleneck. To help this process, we developed a small Scala library optimized for specifying and documenting an XML schema with minimal effort in a way that automatically generates the corresponding parser. This can serve as a good interface between us and the implementer of the system-near export. An exception was Isabelle: because its integrated export code is written in the same language as MMT (Scala), we could skip the generation of files and hand over the data in memory.

A subtle practical issue we encountered with kernel-generated files in multiple situations (e.g., PVS/NASA) is a reliance on case-sensitive file systems. Because logical names declared inside source files are usually case-sensitive, care is needed when generating a separate file for every such declaration: developers must encode the names to make the generated file names unique without relying on case. Otherwise, exports may be entirely unusable on some systems, e.g., when cloning a git repository on a case-insensitive file system, the second file overrides the first, git detects a change, and a subsequent pull fails.

2.4 User-Level vs. Kernel-Level Language

The main drawback of kernel-based exports is that the *user-level* data structures in the source can be very different from the *kernel-level* data structures that make up the export. This is because most systems employ complex processing chains that translate the user-level into kernel-level data structures. This chain includes a varied and growing set of transformations including disambiguation, type inference, proof completion, normalization, and feature elimination, all of which we group together under the term *elaboration*.

Disambiguation includes resolving user-level unqualified or partially qualified identifiers into kernel-level qualified ones, as well as parsing user-level notations into kernel-level abstract syntax trees. Both cases may be ambiguous, and systems usually employ idiosyncratic resolution mechanisms. Disambiguation can be intertwined with type inference, e.g., if the identifiers have the flavor of selectors, the

type or type class of an object o may be used to disambiguate a selector applied to o, or the resolution of the selector may be used to infer the type of o.

Inference adds information to the kernel-level syntax tree that the user-level language omitted because it was inferable from the context. In the simplest case, the inferred information is uniquely determined, most importantly for omitted types of bound variables, implicit arguments, and implicit type coercions. Some systems also employ heuristic algorithms that users can guide with annotations to infer non-unique information. Examples are variable name reservations to infer the type of a variable from its name and Coq's canonical structures to infer a record from a single field.

Proof completion involves applying proof steps, tactic processing, and free proof search. All have in common that the user-level language includes hints that are usually just explicit enough to help the system find a proof. Kernel-level language may represent only the proof hints and throw away the proof after finding it (typical for LCF style systems), or it may throw away the hints and only represent the found proof (typical for proofs-as-terms style systems). If it represents both, it can be difficult to cross-reference the hints to the proof.

Typical examples for *normalization* are definition expansion, β-reduction, and computation of inductive functions. Normalization is always needed as an ephemeral operation during verification. But depending on the system, some normalization steps are permanent in the sense that the kernel-level language only represents the normalized terms. An extreme example is Mizar's permanent normalization of implication into a disjunction.

Finally, *feature elimination* expands high-level derived declarations into low-level primitive ones. Examples include many diverse user-level language features such as records in Coq, structures in Mizar, or inductive types in Isabelle. This kind of elaboration can be especially severe as the resulting kernel-level language tree may be unrecognizably different from the user-level language for anybody who does not know how the elimination is defined.

We describe several issues pertaining to elaboration that come up frequently in exports.

Almost all elaboration steps are not injective, and the information of what the user originally wrote is often lost along the way. Thus, a lot of high-level structure is elaborated away on its way to the kernel, and it is extremely difficult or even impossible to recover this information in an export. But that structure would be extremely valuable for virtually any application other than proof-checking. For example, most systems provide some support for common high-level features such as mathematical structures (modules, records, etc.) or term languages (inductive, co-inductive types), and when translating between systems, it is highly desirable to match these up with each other. In the common situation where different systems eliminate these features in different ways, this is not possible. Exports that preserve this user-level structure remain an open problem for all systems. For example, we only had the resources to preserve Coq sections and Isabelle locales but none of the many other high-level constructs.

Advanced user-level declarations such as inductive types or recursive function definitions are typically justified by meta-theorems that establish their conservativity. The meta-proof may be used explicitly in the system by elaborating user-level declarations (as in Isabelle) or be left to the literature about the system (as

when adding inductive types to the calculus of constructions in Coq). Both can be problematic: The former can lead to more numerous and less intuitive declarations in the kernel, which can cause a strong disconnect between user and kernel-level declarations. The latter may cause subtle theoretical issues (in fact so subtle that discussions usually stay limited to small circles of experts) when the meta-theorems do not perfectly match the implementation, e.g., if the conservativity of two features is proved individually relative to the base calculus without noticing that their combination in the implementation may violate conservativity.

The representation of user-level features in logical frameworks is much harder and less well-studied than the representation of kernel-level languages. That is partially because kernel-level syntax is often defined using the highly standardized methods of context-free grammars and context-sensitive inference systems, which yields a strong relation between specification and implementation. The elaboration of user-level syntax, however, often uses relatively unconstrained programs in Turing-complete programming languages, which is much harder to represent in logical frameworks. MMT provides two mechanisms for representing elaboration-based features. Firstly, *declaration patterns* [23,25] are used for statements whose elaboration can be defined declaratively by simple rules. We use it to define HOL type definitions and the various definition principles of Mizar. Secondly and more generally, *arbitrary elaboration* can be defined in the Scala language underlying MMT. We use this to define a few advanced features such as PVS includes and Coq sections, and we could also use it to capture Isabelle's heavily-used corresponding mechanism for arbitrary elaboration. We recently described details and collected more examples in [45].

A related problem is the destruction of *structure-sharing*. Many systems internally employ sophisticated structure-sharing to reduce memory requirements. Naive exports often cannot reuse this structure-sharing easily, potentially leading to enormous blowups in the generated intermediate files and ultimately in the export. For example, Isabelle exports of the entire AFP that include all proof terms have not been done so far at all because of this explosion. New research into structure-sharing-preserving exports is needed to make proof exports for Isabelle possible. Apart from the practical difficulties, even the theoretical design of such exports is more difficult than it may appear at first. For example, Coq's structure sharing is syntactic rather than semantic, i.e., Coq shares two terms with free occurrences of variables even if they refer to different variable declarations. It is not obvious how to treat this best in an export.

2.5 Exporting Proof Objects

The conflict between user-level and kernel-level is particularly critical for the export of **proofs**. It is still unclear what the best way to export proofs is. We distinguish three cases.

Firstly, the export of *kernel-level proofs* is often straightforward, but the proof objects become huge. This is particularly severe when proofs include automatically found parts, which may be much larger than necessary. For example, our Coq export runs out of memory on some very large proofs in the Feit-Thompson proof [19]. We speculate this might be because an unnecessarily large proof term is built internally. But a detailed investigation has so far been precluded by a lack

of resources. Moreover, kernel-level proofs have only limited value, independent proof checking essentially being their only use.

Secondly, the *user-level proofs* are much more interesting for, e.g., viewing, searching, reusing, or translating libraries, but they can usually not be exported or only be exported in source-near syntax (strings in the worst case) that lack the information inferred by the prover.

Thirdly, *dependency-only* use an MMT expression $dependsOn(d_1, \ldots, d_n)$ as the proof that records only the dependencies that were used. This is sufficient for some applications, such as change management and premise selection as done in [32]. In the simplest case, each d_i is an axiom or theorem identifier, but it can be any expression of the prover's logic, e.g., the application of a polymorphic theorem to the type arguments with which it was used in the proof. Thus, the same type system that is used for the logical data can be used for logical fragments inside the extra-logical proof data.

In any case, kernel-level proofs may still have logical gaps making them insufficient even for proof checking, e.g., if automated proving or decision procedures are part of the kernel (as for Mizar and PVS). They may also have pragmatic gaps if the logic includes powerful automatic computation (as for Coq) as opposed to logics that record complex computations in explicit rewrite steps (as for Isabelle). Depending on the application, these gaps may be seen as advantages (skipping low-level steps that can be easily recreated after translating the proof to another system) or disadvantages (precluding the rechecking of the proofs).

In general, a major lesson of our exports is that the community has not yet converged on a good solution for exporting mid-level proofs that combines the relevant structure of user-level proofs with the inferred information and the re-checkability of kernel-level proofs. In particular, it would be great if applications processing proofs could gradually choose the level at which a subproof is seen. That way a proof translation tool could elaborate a high-level proof only until it encounters mid-level proof steps for which the target system provides a counterpart.

Because of this lack of convergence, all of our exports have de-emphasized proof export and usually opted for dependency-only exports. We only experimented with a few kernel-level proofs for Coq to test the scalability of the approach.

2.6 Heterogeneity

Most proof assistants use what we call the *homogeneous* method, which fixes one foundational logic and builds on it using conservative extensions such as definitions and theorems. Thus, all domain knowledge is ultimately represented in terms of the same primitive concepts. We speak of the *heterogeneous* method if systems define theories that encapsulate choices of primitive concepts and then consider truth relative to a theory. The homogeneous method can be seen as the special case where a single theory is fixed, e.g. the underlying set theory in Mizar or the theory of an infinite type in HOL Light.

The heterogeneous method allows capturing the mathematical maxim of stating every result in the weakest possible theory and moving results between theories in a truth-preserving way. (The formal tool to capture this moving operation are theory morphisms.) This is often called the **little theories approach** [15] as pioneered by IMPS and supported by Isabelle and PVS. Even if heterogeneous

reasoning is possible, it may be **optional**, e.g., users of Coq and Isabelle often do not make use of heterogeneity even when they could. PVS and IMPS on the other hand force users to state results inside theories, and IMPS strongly encourages the use of theory morphisms.

Almost all systems support the heterogeneous method in some way because it is needed at least to capture mathematical structures, such as groups or topological spaces, which are naturally represented as theories (i.e., sets of operations and axioms). In this context, a critical distinction is whether the heterogeneity is external or internal (which we discussed in [44]).

We speak of **internal** heterogeneity if the base logic allows expressing a theory as a type (or a similar construct) in a way that allows arbitrary quantification over all models of the theory. Internal heterogeneity is usually realized via record types such as in PVS and Coq. Mizar structures behave similarly. Internal heterogeneity is not always an option. For example, higher-order logic with record types can only express theories without axioms as record types because record types typically do not support fields for axioms; dependent type theory does not have that limitation.

We speak of **external** heterogeneity if it is a language feature that sits on top of the base logic and provides a way to specify theories. It can be added relatively easily to virtually every base logic (whereas adding internal heterogeneity may be impossible or difficult) and is present in most systems, e.g., Coq modules, OCaml modules for HOL Light, IMPS theories, Isabelle theories, locales, and type classes, and PVS theories. Moreover, a limited form of binding variables typed by a theory is often possible, e.g., via functors in Coq and HOL Light, parametric theories in PVS, or type class ascriptions in Isabelle.

This distinction is important for library integration. Internal heterogeneity tends to be more flexible and more expressive, especially in large projects involving mathematical structures. A good experiment here is the Coq library where both internal and external features are well-supported and in competition with each other: We see that the external module system is falling out of use somewhat and the internal record types are becoming the predominant source of heterogeneity, most prominently in the Mathematical Components project [17]).

But external heterogeneity is easier for a system to provide and makes porting libraries easier. Translations between different base logics are often prohibitively difficult, whereas translations between corresponding external heterogeneity features can be much simpler. In particular, we can export a system's external heterogeneity as Mmt theories and then easily translate those into other systems. For example, translating from calculus of constructions to higher-order logic is very hard, but translating from the theory of natural numbers in the former to the theory of natural numbers in the latter is much easier. On the contrary, libraries written using internal heterogeneity are much harder to port unless the target system supports internal heterogeneity as well.

Therefore, from the perspective of library integration, it is critical to preserve external heterogeneity in the exports. We were able to do that directly for PVS and IMPS. For Isabelle, heterogeneity only exists at the user-level, and elaboration reduces it to a homogeneous kernel-level formalization. But we were able to reconstruct the user-level locale structure and export it in addition. We also preserved the corresponding Coq module structure, but this has limited practical value as it is not widely used throughout the libraries.

2.7 Toplevel Language Extensions

Most provers allow more language features at the toplevel than the logic they are based on.

Processing instructions are extra-logical declarations that control the system behavior down the line. This may even involve side-effectful instructions, whose semantics depends on the prover's linear processing of the source files. The most important application is to guide otherwise inefficient search algorithms. For example, a source file might contain a line that switches on a search heuristic that is then used in subsequent declarations. These instructions may depend on not widely known implementation details, be under-documented, or even be open to user extension. Examples for the latter are users writing a Coq plugin or inlining ML code in Isabelle files.

A common application is to complement undirected proof search with dedicated algorithms for specific subproblems. For example, equational theorems in Horn form can be used for an optimized simplification procedure via rewriting. But usually only some of those theorems should be used at the same time to avoid infinite looping. Therefore, systems like Isabelle and PVS maintain a set of theorems that are used for automatic rewriting, and users can flexibly add or remove theorems to this set. A related example is type-checking in softly typed systems, where type-checking is reduced to proof search. Mizar registrations are special theorems about typing that are used for automated type checking. Similarly, unification hints such as Coq's canonical structures guide the type inference algorithm when having to choose a record if only some it its fields are known.

Implicit toplevel binding allows implicitly or specially bound variables that are treated as if they were universally quantified at the beginning of the containing statement. Characteristically, the respective logic does not feature the corresponding universal quantifier; thus, these variables can never be bound in subexpressions, and their implicit binding at the toplevel of a statement is the only way to introduce them. Examples are type variables in HOL, type and subtype variables in PVS, and function and predicate variables in Mizar. Internally, universe variables in Coq behave accordingly although no user-level syntax is provided.

Technically, adding such toplevel binding changes the base logic. However, as a general rule, this is conservative over the base logic: toplevel binding can be eliminated in favor of all (i.e., usually infinitely many) ground substitution instances. This is well-known from axiom schemas in axiomatic set theory: in order to stay within first-order logic, authors often state a single second-order axiom as a first-order axiom schema with infinitely many first-order substitution instances. Interestingly, when defining the logics in an LF-like logical framework, the formalization of these toplevel binders requires no additional work: they are directly represented using the Π-binder of LF. This can be seen as a rigorous argument for why this toplevel binding is in fact conservative.

While this is a relatively minor theoretical observation, we mention it here explicitly because it is not widely known in all its generality. Therefore, the details in individual provers are not always cleanly documented or implemented, and their treatment can occasionally be confusing when writing exports.

2.8 Non-Logical Information

There are a number semantically irrelevant language features that are often critical to export as well. In some cases, it is even strongly advisable to prioritize them higher than some logical features.

By far the most important non-logical aspect of an export is attaching **source references**. These annotate each element in the export with the corresponding physical location (if any) in the source (i.e., file, line, column). Even though some form of source references is needed anyway for provers to report useful error messages, it is not always easy to preserve them in the export, depending on how they are processed and stored internally. This can be especially difficult if elaboration substantially changes the syntax tree. But source references are crucial to enable many applications of the exports because they allow pointing users to the human-readable sources whenever they interact with a part of the export, especially in the typical case where the kernel-level data structures in the export look entirely unfamiliar to the user. A typical example is search, where one wants to search through the export but show results in the source. Source references should ideally be present *everywhere*, i.e., for each subexpression, but are at least needed for every *statement* (definition, theorem etc.). In the worst case, source references for statements can be *recoverable* by parsing the source in a separate process.

Another obviously desirable non-logical feature are **comments**. Exporting comments is relatively straightforward iff they are preserved during elaboration. MMT can even represent alternated nesting of formal and informal text. But elaboration often does not preserve comments — a common approach to build systems is to discard comments very early, even during parsing. Isabelle is special here in that it includes structured markdown-like syntax for informal text at the same level as formal developments. As a part of the kernel data structures, these are easy to preserve in the export.

Finally, semantic web-style **ontological abstractions** have proved very successful in many areas such as biology and medicine. The analogues for proof assistant libraries systematically abstract away all symbolic expressions (types, formulas, proofs, etc.) and only retain identifiers (of modules, statements, etc.) and properties and relations on them such as authorship, dependency, or check time. While these do not capture the entire semantics of the library, they are extremely powerful for certain problems: Their level of abstraction is sufficient for shallow services such as semantic navigation, search, or querying. Even better, at this level, the huge formal differences between the libraries disappear and services can more easily span multiple libraries, e.g., to find related formalizations in different libraries. Moreover, standardized formalisms such as RDF [58] and SPARQL [64] and highly scalable tools are readily available. While such applications were already envisioned years ago, e.g., in [3,5], they could so far not be realized at large scales because provers were unable to export the necessary data. Therefore, we have recently carried out two major case studies in supplementing the exports described so far with exports of ontological abstractions: in [11], we define an upper library ontology and use it to export linked data representations for the Isabelle (40M RDF triples) and Coq (12M RDF triples) libraries. This enables, for example, querying for all Isabelle or Coq theorems about a given mathematical concept. Corresponding extensions of the other exports are straightforward conceptually; while they

still require a considerable investment in collaboration with the system expert, they are significantly easier than full logical exports.

2.9 Library Structure

Due to the tremendous logistical challenges in maintaining large libraries with many authors over decades, multiple models for the **structure of libraries** have been developed. Typically each system is *integrated* with a standard library that is co-released with the system, often maintained in the same repository as the system's source code. This library can be comprehensive or small and be extended by any number of external *distributed* libraries — HOL Light with its integrated library mostly curated by the main developer and Coq with many distributed libraries developed by users can be seen as opposite extremes on this spectrum.

For external libraries, a critical question is whether these are *co-maintained* with the system, usually as part of the regression test suite that is checked before every system release. In that case, the library can be a collection of independent user-*submissions* or a single curated *coherent* body — Isabelle/AFP with many relatively unrelated submissions and the Mizar library carefully curated by a small committee are opposite extremes on this spectrum. Both are officially co-maintained with the system. PVS/NASA is special in that it is a single external library that is developed independently from the prover; in practice, it is used as the main regression suite for the system.

Notably, if many distributed libraries exist (as for Coq) or if the central library is submission-based (as for Isabelle), there is no guarantee that all developments are compatible with each other. They may depend on different system settings (e.g., impredicativity for Coq) or system states (e.g., loaded language extensions or automations), or try to register the same names. Therefore, it is not always possible to speak of *the* system export; even within a single library, different parts may be inconsistent with each other.

3 The Coq Libraries

Coq is based on constructive type theory and offers strong support for general purpose (pure) programming and theorem proving. It supports a diverse user community large enough to require decentralized library maintenance and hosts multiple flagship projects from mathematics (such as the Odd Order proof [19]) and computer science (such as the CompCert C compiler [40]).

Our Coq export was carried out together with Dennis Müller and Claudio Sacerdoti Coen. The details were published in [59,46].

3.1 Language

Overview and Formalization Coq is based on the calculus of inductive constructions (CIC), which can be roughly seen as dependent type theory plus universe hierarchy plus (co)inductive types. The exact status of propositions is subtle, but essentially every proposition is a type. Generally, the most desirable representation in a logical

	Coq	HOL Light	Isabelle
foundation	dep. type theory	HOL	intuitionistic HOL
internal heterog.	records	none	records, no axioms
external heterog.	modules	OCaml modules	locales
theory morphisms	-	-	+
library organization	distributed	integrated	submission-based
co-released	-	+	+
exported	70/250 libs	std library	std library, AFP
system-near export	instrumenting	instrumenting	traversing
status	plugin	modification	integrated
format	XML	OMDoc	in memory
# loc in prover	major part	1k	major part
# loc read to MMT	2k	-	2k
# MMT decls. in export	$170k$	$20k$	$2M$
# RDF triples in export	$12M$	N/A	$40M$
proof export	low-level	low-level	low-level
coverage	full	dependency-only	dependency-only
not included	computation	-	-
src refs	statements	recoverable	everywhere
collaborator	Sacerdoti Coen	Kaliszyk	Wenzel
reference	[46]	[31]	[38]

	IMPS	Mizar	PVS
foundation	HOL+X	set theory	HOL + X
internal heterog.	none	structures	records, no axioms
external heterog.	theories	none	theories
theory morphisms	+	-	+
library organization	integrated	curated	distrib, co-maint.
co-released	+	+	+
exported	std lib	MML	std lib, NASA lib
system-near export	traversing	traversing	traversing
status	modification	integrated	integrated
format	JSON	XML	XML
# loc in prover	2k	major part	major part
# loc read to MMT	8k	4k	2k
# MMT decls.	$2k$	$70k$	$25k$
# RDF triples	N/A	N/A	N/A
proof export	low-level	high-level	partial
coverage	dependency-only	partial	dependency-only
not included	-	automation	automation
src refs	recoverable	none	everywhere
collaborator	Farmer	Urban	Owre
reference	[42]	[27]	[36]

Table 1 General Aspects of Theorem prover exports

framework is a Church-style typed one: it uses $\texttt{tp} : \texttt{type}$ to represent every Coq-type A as an LF-term $\ulcorner A \urcorner : \texttt{tp}$, and $\texttt{tm} : \texttt{tp} \to \texttt{type}$ to represent every Coq-term $t : A$ as an LF-term $\ulcorner t \urcorner : \texttt{tm} \ulcorner A \urcorner$. This has the advantage that only well-typed terms can be encoded at all, and Coq's typing rules are captured by the LF type system.

But this representation requires more type annotations than typically present. For example, if $A \Rightarrow B$ encodes simple functions, the application operator is represented as $\texttt{apply} : \Pi_{A,B:\texttt{tp}} \texttt{tm}(A \Rightarrow B) \to \texttt{tm}\, A \to \texttt{tm}\, B$. Thus, a Coq application is represented as an LF term that records its input type A and output type B. (Actually, Coq uses a dependent function space, but the same argument applies). In most cases, this additional information is advantageous and is already present

in the kernel data structures or can be inferred relatively easily, by calling either Coq functions during the export or logical framework functions in Mmt. But it significantly increases the size of the export. This size explosion problem can be alleviated partially by using rewriting as in [9]. This allows annotating only Π-expressions with types but not applications. These approaches to size issues so far do not differentiate between terms, types, and proofs, and that might help in the future.

In the case of Coq, we exported low-level proof terms, and a typed representation would not have been feasible yet. We used an untyped Curry-style representation based on a single type $\texttt{expr} : \texttt{type}$ of all Coq terms and types, and a separate typing judgment $\texttt{of} : \texttt{expr} \to \texttt{expr} \to \texttt{type}$ of typing derivations. Now the application operator is simply formalized as $\texttt{apply} : \texttt{expr} \to \texttt{expr} \to \texttt{expr}$ together with appropriate typing rules. However, our export does not preserve Coq's internal structure sharing because it is unclear how to its purely syntactic sharing (where, e.g., even different variables of the same name are shared). This is presumably why even this untyped encoding ran out-of-memory on some of the largest proofs in the Feit-Thompson proof [19].

The untyped encoding also loses the types of Coq-identifiers $c : A$, which simply become $c : \texttt{expr}$ in LF. To remedy this, one can represent $c : A$ using two declarations $c : \texttt{expr}$, $c\_type : \texttt{of}\, c \ulcorner A \urcorner$. To avoid making the encoding non-compositional in this way, we used Mmt's ability to flexibly extend the logical framework: We switched to LF with predicate subtypes, and defined $\texttt{tm}\, A$ as the subtype of $\texttt{expr}$ containing only those $e : \texttt{expr}$ that satisfy $\texttt{of}\, e\, A$. This allows compositionally representing $c : A$ as $c : \texttt{tm} \ulcorner A \urcorner$ as in Church-style encodings.

Universes Coq's universe hierarchy makes the Church representation even more complex because a second parameter is needed: $\texttt{univ} : \texttt{type}$, $\texttt{tp} : \texttt{univ} \to \texttt{type}$, $\texttt{tm} : \Pi_{U:\texttt{univ}}.\texttt{tp}\, U \to \texttt{type}$. The application operator now takes four additional arguments instead of two as in $\texttt{apply} : \Pi_{U,V:\texttt{univ},A:\texttt{tp}\, U,\Pi B:\texttt{tp}\, V} \texttt{tm}\, M\, (A \Rightarrow B) \to \texttt{tm}\, U\, A \to \texttt{tm}\, N\, B$. Moreover, additional formalization steps are needed to compute the universes M and N from U and V according to the Coq typing rules. Finally, as Coq's universe hierarchy is cumulative, the Church representation breaks down entirely as injection functions have to be inserted non-compositionally all over the place to cast types into higher universes. This is an additional reason why only a Curry representation is feasible at this point.

Coq source syntax only contains the identifier `TYPE` without clarifying its universe, and the system internally introduces a fresh universe identifier for which it then infers constraints. Because these constraints are affected by how an identifier is declared and how it is used, this information must be maintained globally. If an identifier is used in multiple different source files or even libraries, the resulting constraints may be inconsistent with each other. Technically, a Coq export must include these universe constraints and treat them as assumptions relative to which the theorems are stated. This is what we did. But as this information is usually not interesting to the user and not portable to other provers, it remains an open question how to handle these constraints at all.

Inductive Types The description above is only accurate if we ignore Coq's (co)inductive types. These are very complex, e.g., using multiple kinds of parameters and offering primitive operators for recursion and pattern-matching. Capturing

the associated typing rules in a declarative logical framework has so far been out of reach, be it Church or Curry-style. And eliminating them (even if we are willing to lose this high-level feature) is a prime example of a theoretically possible but practically doomed non-compositional logical transformation. Therefore, our representation had to go outside LF by using untyped MMT symbols to represent recursion and pattern-matching. Since our representation was untyped anyway, this allowed exporting all Coq expressions regarding (co)inductive terms without significant additional loss. A representation in a stricter framework like LF or Dedukti ($\hat{=}$ LF modulo) would be very difficult.

3.2 System

The Coq system has grown for several decades, and its internal workings are extremely complex. More recent reimplementations of essentially the same language like Matita [4] or Lean [14] have been able to simplify the implementation drastically. But because they are neither binary nor source compatible with Coq (Matita could read Coq binaries at one point but then diverged), they cannot process the huge Coq libraries.

Recently, Coq development has increasingly focused on giving users more control how their input is interpreted before it reaches the kernel. Type classes and canonical structures/unification hints are the most important examples. These are not visible to the kernel and therefore not part of exports like ours.

Similarly, all Coq proof objects are low-level λ-terms, and high-level structure via tactic languages is elaborated away. This makes Coq exports very big if they include proofs. Because computation (in particular, recursive functions on inductive types) is a kernel feature and thus not part of the proof terms at all, some complex proof steps do not have an effect on proof size. Therefore, it was feasible to export all proof terms.

The Coq module system is primitive and fully visible to the kernel. That allows fully preserving the modular structure. However, the trend in Coq development is towards using record types (which are treated as inductive types with one constructor) instead of modules. While visible to the kernel, this modular structure is very hard to recover. Therefore, there is no export yet that can identify this modular structure.

Maybe surprisingly, the Coq system includes several imperative aspects. For example, a section is checked as usual, but when closing a section the internal state of Coq holding the local context of the section is rolled-back and the section is installed in the way in which it is visible from the outside. Thus, the local context of the section no longer exists at all, which makes it impossible to export sections via the kernel-generated binary files and difficult to export via kernel instrumentation. We opted for the latter.

3.3 Libraries

Coq has reached the usage size where the central maintenance of libraries is no longer feasible. Instead, the Coq library has been factored into hundreds of repositories with a somewhat standardized build process. This allows distributed main-

tenance of the library. But it also means that not every repository always builds with the latest version of Coq. For example, when we ran our export in early 2019, only around 70 of around 250 repositories could be built, including the MathComp libraries (the situation has improved since then).

We used the XML files produced by kernel-instrumentation in [59] for our export [46]. Due to the distribution of libraries, significant additional scripting was needed to detect all libraries, identify their metadata and dependencies, and iterate through them. The metadata does not include which toplevel Coq namespace(s) a library's declarations are in so that additional checks were needed to determine which declarations to export.

Alternative exports are presented in [6,9] using Church representations in Dedukti after eliminating (co)inductive types and in [13] to first-order logic for the purpose of premise selection, but none covers the entire language.

3.4 Outlook and Open Challenges

The biggest future challenge for Coq is to **scale up** the export. Our export is deceptively scalable — to the point of including all proof terms — because it uses an untyped representation in the logical framework. Switching the export code to work relative to a typed representation would be straightforward but would yield much bigger exports. The usefulness of the untyped representations is limited because any kind of type inference of re-verification must implement the Coq type system from scratch. The rewriting-based representations in Dedukti may alleviate this problem. But we expect that only a systematic solution to proof export as indicated in Section 2.5 will eventually be successful.

This will require two investments: Firstly, Coq must process its proofs in a way that allows recovering **mid-level proof terms** — terms that includes more information than the users tactic script (e.g., all intermediate proof states and tactic invocations) but less than the low-level λ-terms. Secondly, proofs that are the result of search and other computations must be refactored to ensure they are not massively larger than necessary. That would also help cause of program extraction — it is plausible that some programmatically found current Coq proofs are so convoluted that they would not allow for extracting elegant programs. As a simple immediate step, it may help for Coq to display the internal size and possibly structure of every proof. That would allow users to notice when proofs are much bigger than expected.

The second huge challenge, which is also a requirement for a **typed representation**, is to define a declarative representation of the Coq typing rules in a logical framework. Here, the treatment of (co)inductive types and recursion is currently not possible with state of the art frameworks. We circumvent this issue by using untyped representations; the exports in Dedukti circumvent it by using a typed representation that does not cover these features and applying a non-compositional transformation to eliminate them during the export. We expect building appropriate frameworks and applying them to Coq will be an effort measurable in person years (even when using Mmt for rapid prototyping). This would ultimately also require sorting out a few subtleties in the inner workings of Coq such as the treatment of record projections (which are primitive in the kernel for efficiency even

though records in general are not) or the various corner cases in the handling of inductive types.

A major Coq-specific logistical challenge is given by the **distributed library**. While necessary due to the size of the user community, it brings a host of new maintenance problems such as different libraries depending on different versions of Coq. It is likely that this problem will not go away because new versions of Coq will be released faster than all existing content will be adapted. Therefore, new tools will be needed to manage the symptoms. However, this is a general issue facing the Coq community, and we expect good package and build managers to emerge in the near future, which can then be integrated with exports. That will make the scripts obsolete that we used to run our export over all libraries and will ensure the maximal number of libraries can be exported.

Finally, while large scale library integration and system interoperability is mostly a problem of the future, we can already foresee that the predominant use of record types for heterogeneity will cause problems. A good compromise might be to annotate those record types that are meant to be used in the sense of mathematical theories so that exports can translate those records differently. This might require limiting the use of such records: for example, record types can only be mapped to Isabelle locales if they do not occur as the types of non-toplevel bound variables.

4 The HOL Light Library

HOL Light is a minimalist implementation of standard higher-order logic without complex additional features with a small and easy to understand kernel. It is maintained essentially by a single developer and features a large integrated and coherent library. That made it flexible and scalable enough to be chosen for the Flyspeck project [20]. Due to its simplicity, the language can be embedded into most other logics (Mizar, because it lacks λ-abstraction, being the main counter-example). This combination makes it very popular source language for translations [47,33,7,29], and it was the first for which a major library translation was done [50].

Our HOL Light export was carried out together with Cezary Kaliszyk. The details were published in [31].

4.1 Language

We formalized the HOL Light logic in LF using the analogue of the Church encoding sketched in the section on Coq. Because this was the first export we did, we refrained from exporting proof terms at all so that the Church encoding did not present a scalability hazard. Instead, we exported dependency-only proofs.

4.2 System

HOL Light is implemented inside the OCaml toplevel with only a few tweaks for parsing expressions. This makes the kernel very easy to instrument for generating

an export. However, it makes it near impossible to export any of the high-level features used to build the library as these are arbitrary OCaml programs on top of the kernel. For example, the kernel is not even aware of the list of theorems, which are instead maintained by the OCaml context. Any high-level or tactic-based proofs are invisible to the kernel and only low-level exports scale to the whole library. This is particularly unfortunate in the case of HOL Light because the source level proofs are just highly concise tactic invocations and thus can be very hard to read. To remedy this problem, [49] patches individual tactics in order to export tactic applications in addition to kernel-level.

Because the source files can contain arbitrary OCaml code, recovering source references is difficult in principle. However, because the library uses consistent conventions it is relative easy to write parsers that identity the source lines of statements.

HOL Light inherits the OCaml module system. However, this is rarely used in practice. In any case, any modular features are invisible to the kernel.

4.3 Library

The HOL Light library is highly integrated with the system and maintained by the same single person. That makes it very smooth to export by kernel instrumentation. Our export [31] modified the existing instrumentation to output OMDoc files directly. Additionally, it accesses an internal table with notations in order to preserve those in the export.

The most important other library is the Flyspeck proof [20]. We have not exported it because it is extremely big. It would also not be particularly interesting (except for porting the proof itself) as most generally reusable formalizations have been migrated from Flyspeck to the main library already.

4.4 Outlook and Open Challenges

HOL Light is among the easiest systems to export and to maintain an export for. This is due to the simplicity and stability of language, system, and library: The language is straightforward to model in a logical framework, and HOL Light avoids adding advanced or experimental features to HOL. The system is specifically designed as a lightweight rarely-changing kernel, which makes it easy to understand and instrument the kernel to export proofs. And the library is very coherent and uses uniform conventions.

But its low-level kernel that users can feed arbitrarily with programmatically generated proofs makes it extremely difficult to export anything but low-level proof terms. Therefore, we cannot expect naive proof terms exports of large projects such as Flyspeck to scale without systematic changes. Even if a standard for mid-level proof terms in the sense of Section 2.5 exists, it will be difficult to **export proof terms** from HOL Light. The most realistic option would be to instrument not only the kernel but also individual tactics, and efforts in this direction have been made already [49]. But as users can write arbitrary tactics, this will require a significant investment of expert knowledge to be comprehensive. Establishing best practices and special macros for tactics with an eye towards instrumentation is helpful here.

Another major challenge is that HOL Light uses almost exclusively the homogeneous method, which makes it more difficult to integrate with other libraries. Here it would be helpful to research the **automated introduction of heterogeneity** by abstracting from the assumptions theorems and grouping them according to their assumptions.

5 The IMPS Library

IMPS was one of the earliest proof assistants. It used a higher-order logic extended with features for subtyping and partial functions. Despite pioneering some original features that are still of interest for proof assistants today, it has fallen out of use before building a large library. Today it only runs on two machines: one installation by one of the original developers (Farmer), and one of ours for our export. Its inclusion in our set of case studies was motivated not by size of library and user base (as for the other provers). Instead, it was motivated by (i) IMPS's commitment to the heterogeneous method, which will deserve more attention to make future library integrations more feasible, and (ii) to permanently archive its library before the technology to process it dies.

Our IMPS export was carried out together with Bill Farmer and Jones Betzendahl. The details were published in [8].

5.1 Language

IMPS's underlying logic, called LUTINS, is a variant of Andrews' higher-order logic $\mathbf{Q}^0$ [1]. Most characteristically, all functions are (potentially) partial, and terms may be undefined. The latter is captured by a primitive unary predicate for definedness.

IMPS uses a limited subtyping system: underneath each base type, a hierarchy of subtypes (called sorts in IMPS) may be introduced. Due to the partiality, the domain type of a function type is only an upper limit on the set of arguments for which the function is defined. Interestingly, that makes the function type operator covariant in both arguments.

Base types for individuals and binary booleans is built-in, and undefined constant applications that return booleans are considered false. Other base types and their subtypes can be declared in theories. Contrary to most other proof assistants, IMPS systematically uses the heterogeneous method, and theories routinely contain base types and operations on them whose properties are not given by definitions but by axiomatizations.

Some flexibility exists in how to represent LUTINS in LF. It is straightforward to give a Curry-style representation using $\texttt{tp} : \texttt{type}$, $\texttt{tm} : \texttt{type}$, and $\texttt{of} : \texttt{tm} \to \texttt{tp} \to \texttt{type}$. Alternatively, we can give a Church-style representation for the maximal types and then a Curry-style representation for the sorts underneath each type: this would use $\texttt{tp} : \texttt{type}$, $\texttt{tm} : \texttt{tp} \to \texttt{type}$, $\texttt{sort} : \texttt{tp} \to \texttt{type}$, and $\texttt{of} : \Pi_{A:\texttt{tp}} \texttt{tm}\, A \to \texttt{sort}\, A \to \texttt{type}$. The latter allows LF to perform some type-checking automatically at the cost of making the representation a bit clunkier. For example, the latter requires all polymorphic operators to take two arguments: one for the type A and one for the sort $S : \texttt{sort}\, A$. Finally, we can use a complete Church-style

representation using $\texttt{tp} : \texttt{type}$, $\texttt{sort} : \texttt{tp} \to \texttt{type}$, and $\texttt{tm} : \Pi_{A:\texttt{tp}}\texttt{sort}\, A \to \texttt{type}$. This yields the most elegant representation of the typing rules, but it requires explicit casts whenever subsorting is applied. It is relatively easy to switch between these, and the details of the trade off are ongoing work.

The only subtlety is the handling of binders as bound variables always range over defined values: All typing rules that introduce variables (λ-abstraction, universal introduction, existential elimination) must add assumptions to the context that assume the definedness of (the instances of) the bound variable. Correspondingly, the rules that perform substitutions (β-reduction, universal elimination, existential introduction) must have premises that ensure the substituting term is defined. In the first two alternative representations, this can be taken care of by formulating the rules in such a way that $\texttt{of}\, t\, A$ only holds if t is indeed defined.

5.2 System

IMPS is implemented in LISP, and uses a LISP-like source syntax. Because it has not been maintained for years, it is rather difficult to work with. But previous work [42] had already developed a partial instrumentation to generate OMDoc files. Because that work predates Mmt and did not use any rigorous semantics for OMDoc, let alone logical frameworks, it is better seen as system-near XML export rather than as a semantics-preserving export.

We adapted and extended this earlier work and developed into a system-near export in JSON. Because the export was not used internally by IMPS, this was a harmless modification of the system. The IMPS parser does not record comments or source references, and to recover those, we wrote a second parser for the IMPS source syntax from scratch (in Scala): it parses just enough of the IMPS syntax to find the names of declarations and where they are declared in order to produce source references for declarations. Eventually, we wrote a Scala program to merge the two data structures to build Mmt data structures in Scala, which could be easily serialized.

IMPS stores proof objects that record tactic invocations made through a graphical interface. In addition to user-level tactics, special tactics (called Macetes in IMPS) are automatically generated from theorems of certain shapes, e.g., to turn theorems in Horn form into proof rules. This is particularly important to automate the tedious reasoning about subtyping and definedness. These proof objects contain gaps where IMPS automation was used. Thus, the proofs are already roughly mid-level proofs that can be more easily translated to other provers, and we expect that the missing steps can be recreated by the automation in those provers relatively easily.

5.3 Library

Because the library is small, frozen, and part of the IMPS source code, it is relatively easy to export it once and for all in a single run.

Only two subtleties arose in the structuring mechanisms used by IMPS. Firstly, IMPS theories are split into languages and theories and organized via theory en-

sembles. But that idiosyncrasy is arguably less critical to preserve, and we represented all as Mmt theories.

Secondly, IMPS's treatment of theory morphisms includes a subtle imperative part: after defining a theory T and some theory morphism $m : S \to T$, users can install the morphism by imperatively adding the translations of S-theorems to T. While straightforward in an implementation that imperatively maintains theories as tables of declarations, it is problematic in exports. We could mimic it Mmt only by defining a new theory T_m that conservatively extends T with the corresponding definitions.

Finally, for a few rarely used unusual features such as the definition of what IMPS calls quasi-constructors, we simply collected all instances and formalized them in LF manually.

5.4 Outlook and Open Challenges

Our export provides an archive of the IMPS library in OMDoc. Additionally, because we also built system-near JSON representations and a fresh parser of the IMPS library, other exports can be built off these IMPS-near data structures in a way that by-passes OMDoc.

Even though IMPS is not in active use anymore, our export is designed to be maintainable so that future extensions can be developed. This would focus in particular on the export of mid-level proofs and on improvements to the representation of the idiosyncratic parts of the type system such as undefinedness and subtyping.

6 The Isabelle Library and the AFP

Isabelle was developed as a logical framework [52] focusing on automated theorem proving. Its standard library includes multiple logic definitions and developments for them, most notably Isabelle/HOL and Isabelle/ZF. Isabelle/HOL is the most visible Isabelle logic nowadays and is used, e.g., in the L4 verification [34]. Isabelle is extremely user-friendly with an out-of-the-box installer and powerful graphical user interface. It is used widely enough to counter-indicate an integrated library, but formalizations are collected by submission to the Archive of Formal Proof.

Our Isabelle export was carried out together with Makarius Wenzel. The details are published in [38].

6.1 Language

Isabelle uses a very simple higher-order logic called Pure as a logical framework. Therefore, we did not formalize Pure in LF; instead, we defined Pure as a logical framework in Mmt, i.e., as a sibling to LF.

Concretely, we represented Pure as the extension of LF with a type of propositions and shallow polymorphism. The latter allows using LF's Π-binder also for type variables at the toplevel, which corresponds to the polymorphism of Pure.

The primitive connectives of Isabelle (equality, universal quantification, and implication) are already part of the automated export because they are themselves declared in the theory Pure in Isabelle's standard library. Extending LF is overkill as Pure is simply-typed, not dependently-typed, but an implementation of polymorphic LF was already available in Mmt. The additional expressivity has no downsides and is in fact helpful because we use the dependent types to represent Isabelle proofs as terms.

Because our export works at the level of Isabelle, it copies over all logic definitions in the Isabelle standard library as they are. In particular, as the types of Isabelle/HOL are a fragment of the types of Pure without using a Church or Curry encoding, both Pure and HOL types and functions are exported as types and functions of the logical framework. This eliminates the scalability hazard we discussed for Coq.

Isabelle features a complex module system with locales (and as a special case type classes) and morphisms between them. Their foundational details are very subtle and not always obvious, but these features are elaborated away almost entirely by Isabelle. Our export stays very close to the kernel-level representation, which is relatively simple to export and still recognizably similar to the user-level structure. While we usually advocate retaining the user-level structure as much as possible, the elaboration-based export has advantages here because Isabelle's kernel-level language already regularizes some of the intricate idiosyncrasies of its user-level language. For example, we did not have to worry about the handling of type variables and contexts.

In particular, our export produces one predicate P_l for every locale l, which takes the types and operations of l as inputs and returns the conjunction of the axioms of l. All constants, definitions, and theorems of l are exported as constants that similarly abstract over all types and operations; additionally, theorems of l are relativized by P_l.

6.2 System

Isabelle provides a heavily-used mechanism for users to define their own high-level declarations, whose semantics is defined by elaboration into more primitive one. Each one of those is called a specification construct. Examples include inductive types and recursive function definitions. These specification constructs are invisible to the kernel and lost during elaboration.

Mmt provides a corresponding mechanism for user-defined high-level features, which we could in principle use to preserve specification constructs features in the export. However, as elaboration is implemented directly in ML, this process cannot be automated, and a manual effort is needed for each specification construct to (i) define the corresponding feature in Mmt and (ii) extend the export with code for detecting and exporting its instances.

Similarly, the locale structure is lost entirely in the direct export. In particular, elaboration simulates the external heterogeneity of the user-level languages in terms of internal heterogeneity in the kernel. To recover the external heterogeneity, we have additionally reconstructed the original locale definitions and exported them as a diagram of Mmt theories. For example, a sublocale l and a locale l' become two theories T_l and $T_{l'}$ with a morphism between them. The export is

redundant in that it contains both these theories T_l and the predicates P_l produced by elaboration. While T_l covers all aspects of the definitions of a locale in a way close to user-level language, all uses of a locale are currently exported in terms of P_l. A deeper integration between T_l and P_l remains future work.

6.3 Libraries

Isabelle uses a coherent co-maintained standard library, which introduces in particular Isabelle/ZF and Isabelle/HOL. Additionally, user-submitted contributions are collected in the Archive of Formal Proofs library and, once accepted, also co-maintained.

Isabelle is the easiest to work with of all the systems we discuss here, featuring easy installation and a high-level export interface via Isabelle/PIDE. Our export work [38] is based on and helped facilitate major upgrades to the PIDE infrastructure, which has led to a very smooth kernel-instrumenting export module that is fully integrated with Isabelle. Moreover, because PIDE and Mmt are both written in Scala, we could skip the generation of intermediate files: we built an Isabelle plugin that makes the Mmt library available so that Isabelle can directly pass Mmt data structures to Mmt. This allows producing OMDoc files directly from Isabelle, making it the most maintainable and scalable of our exports.

However, even with the more space-efficient representation of expressions, Isabelle proof objects become very big. This is because (i) many commonly-used types (in particular inductive and record types) are elaborated, (ii) Isabelle's strong integration of automated provers makes it easy for users to build large proofs, and (iii) the lack of implicit computation requires elaborating all computations into equational reasoning. Therefore, we have opted for a dependency-only export of proofs.

6.4 Outlook and Open Challenges

The Isabelle export is extremely well-maintainable as most of it was written by a core developer and deeply integrated with the Isabelle system and code base. The two biggest future challenges will be avoiding elaboration and reducing the size of proofs terms.

Regarding **elaboration**, Isabelle is special among proof assistants in that it uses a very complex processing pipeline from heterogeneous user-level declarations to homogeneous kernel-level declarations. Not only does any export based on instrumenting the kernel lose a lot of valuable structure, it is also very difficult to modify the system to preserve that structure. The most realistic approach is to (i) modify the Isabelle data structures for user-level declarations in such a way that they must also govern how their concrete instances can be exported. For example, the feature for inductive types would have to include a function e for exporting any concrete inductive type definition. Existing language features would then (ii) have to be adapted to this new interface, e.g., by adding that function e. This could then be combined with (iii) mimicking the user-level features in Mmt to obtain a structure-preserving export. While (i) and (iii) are relatively easy, (ii) requires understanding and adapting a lot of code for the individual features — something

that even experienced Isabelle kernel developers may not be able to do easily. For example, the developers of the inductive type feature themselves might have to write the function e, and doing that even for just the most important features would be a major effort.

Regarding the size of **proofs**, we expect the problem to be even bigger than for Coq because the computation of inductive functions (which is a primitive kernel feature in Coq) is realized via equality reasoning is Isabelle and therefore explicitly part of the proof term. Moreover, Isabelle's extremely strong integration of automated provers increases the disconnect between user-level and kernel-level proof size: user-level proofs may simply consist of invoking automation, which then may produce highly circuitous proofs. In particular, the automatically found proof may differ strongly in size and structure from the proof a human would write or want. This can be alleviated somewhat by engineering efforts such as enabling structure sharing in the export. But to obtain an export of mid-level proofs as discussed in Section 2.5, investments into proof simplification may be needed. On the positive side, Isabelle's Isar language already allows users to write mid-level proof terms explicitly. Thus, it is promising (but still difficult) to export Isar proof terms enriched with information from kernel-level proofs.

7 The Mizar Mathematical Library

Mizar is one of the oldest theorem provers and the only major one to be initiated on the Eastern side of the Iron Curtain. It is set apart from most other provers in multiple ways, which reinforce each other: It focuses almost exclusively on mathematical content (indeed, it lacks features like inductive types or λ-abstractions, which are commonly used in computer science and present in most other proof assistants) and uses first-order set theory as the base logic with a highly idiosyncratic syntax. It is supported by a relatively small developer and user community that has experienced comparatively little cross-pollination with other prover communities. Yet, it features a large centrally curated library, and many of its seemingly unusual features have stood the test of time for formalizing mathematics. Papers on formalizations are generated automatically from Mizar sources and published in a special journal.

Our Mizar export was carried out together with Josef Urban and Mihnea Iancu. The details were published in [27,26].

7.1 Language

Due to its idiosyncratic syntax, the Mizar language is often confusing. But logically, it is actually very simple: it is a formalization of Tarski-Grothendieck set theory in untyped first-order logic with toplevel second-order binders (which are needed to formalize schemas). This can be formalized routinely in LF. Structures/records are realized as functions, whose domain is a special set of names. This is also easy to formalize although, without Mizar's special syntax support for it, the representation is hard to read. Mizar also allows for a large but fixed set of definition principles. These were difficult to identify and compile, but then straightforward to represent as high-level Mmt statements [26].

Additionally, Mizar uses a soft type system, where types are defined as unary predicates over sets. This type system is undecidable, and Mizar provides automation for discharging type-checking obligations, partially guided by user-tagged typing theorems (called registrations). Because this type system is internally represented as typing axioms/theorems about the terms in untyped first-order logic, the direct representation in LF is also straightforward.

However, we could do better: Our Mizar export predates Mmt implementations of LF with predicate subtypes. In a reimplementation, we would investigate representing a Mizar type T as the predicate subtype of the type of sets containing those sets that satisfy $\ulcorner T \urcorner$ (akin to how we used predicate subtypes in the Coq export). This would allow preserving the type system in the export. However, in that case, rechecking the export would require reproving the type-checking conditions discharged by Mizar, a non-trivial task.

7.2 System

Mizar was originally designed in relative isolation. Moreover, due to its focus on mathematics, the user community has remained much smaller than those of the other systems that can be used for software verification. Therefore, it is the least accessible of all the provers we worked with.

A Mizar export can be best written based on the XML files generated by Mizar. In fact, these were introduced by Josef Urban for that purpose [63] and later became used internally. Therefore, they contain all relevant information. But the XML schema is very complex, not always documented, and may change across Mizar versions.

7.3 Library

Our export [27] uses the XML files. It was relatively easy to build and to export the Mizar Mathematical Library, the single coherent library that is co-maintained with Mizar. The main difficulty was that a system expert was needed to explain the XML schema: because the XML is used internally, it also represents many implementation details that are irrelevant for the export and is therefore even more complex than the user-level language.

The internally used XML is undergoing regular improvements, partially in response to our feedback. While this has generally made the structure of the XML better and the export easier, this has made the export very expensive and near impossible to maintain over multiple years.

Mizar uses a big kernel with substantial built-in automation, in particular for the soft type system. These proof steps are not exported and therefore occur as omitted steps in the export. Except for these gaps, the proofs are mostly available in the same high-level format in which they were written by the user.

7.4 Outlook and Open Challenges

Maintainability is a big challenge for our and future Mizar exports. The concepts of the Mizar language, its code base, and the structure of the generated XML

files are hard to understand. This is partially inherent in the system and partially an artefact of the Mizar community overlapping less with over proof assistant communities and the broader computer science community.

A logistical challenge here is that the Mizar developer community is small and mostly concentrated in Białystok, Poland, and therefore not as deeply connected with other prover communities. Moreover, the resource-intensive maintenance of the system and the library limits the community's ability to invest into prover-near exports. But recent improvements to the Mizar XML representation by the Mizar developers are very promising, and we have recently resumed our export efforts in collaboration with Artur Kornilowicz.

A promising avenue is the **reconstruction of the Mizar system in a logical framework** in such a way that the Mizar library can be migrated to the framework and — in the long run — the Mizar system retired. The main challenges here are mimicking the highly human-friendly concrete syntax of Mizar and the automated reasoning capabilities. Promising first results have been accomplished in [30] on top of Isabelle. A modern reimplementation on top of an LF-like framework, e.g., implemented in Mmt, could also be promising: it would allow for supporting Mizar's dependent types more naturally but would hardly be able to match Isabelle's automated reasoning.

A **proof export** from Mizar is generally difficult because not all proofs are currently stored in the system. Because the Mizar kernel is large and complex, instrumenting it for full proof export would require a major effort. However, an export of mid-level proofs in the sense of Section 2.5 is already possible although we did not focus on it. A related problem is that Mizar elaboration expands many abbreviations including for basic symbols like implication. Both instrumenting these expansions during or reintroducing the abbreviation after an export are very difficult, and only a few qualified system experts exist for such tasks. A reconstruction inside Isabelle would allow reducing those problems to the existing Isabelle export technology.

8 The PVS Prelude and the NASA Library

PVS is a proof assistant based on higher-order logic with powerful additional features inspired by mathematics (classical logic, subtypes), the heterogeneous method (theories and morphisms), and proof automation (in particular decision procedures). Using an unusual base logic and being developed by a relatively small group of people at a Californian research institute, its user community and publication footprint are smaller than and overlap less with those of HOL Light, Isabelle, and Coq. But this is not indicative of its quality, and it is used in many research projects, most prominently by NASA, which maintains the largest PVS library.

Our PVS export was carried out together with Sam Owre, Natarajan Shankar, and Dennis Müller. The details were published in [36].

8.1 Language

PVS uses higher-order logic for which a Church-encoding works well. But it adds major additional features that go beyond what can be formalized in any declar-

ative logical framework. Therefore, we extended LF with appropriate features as described below.

Type System PVS uses predicate subtypes. Thus, the typing relation is undecidable and therefore cannot be represented in a Church-encoding in a framework with decidable typing. Similarly, it introduces subtyping, which is highly impractical to represent in a Church encoding. It would require a non-compositional transformation that introduces names for every *occurrence* of a predicate subtype (applying closures if within the scope of binders) and inserts a named injection functions for every application of subtyping. And because it is undecidable whether tho predicate subtypes are equal, these injections would have to be generated gradually whenever PVS proves a subtype relation.

Instead, we extended LF with predicate subtypes (which we already used for Coq already). This makes it possible to formalize PVS's predicate subtypes in a straightforward way while maintaining the advantages of using a Church encoding. We inferred the additional arguments required for the Church representation in the logical framework, which turned out to be just feasible for a dependency-only export of the PVS libraries. Alternatively, we could have used the weaker Curry-encoding as for Coq.

Module System PVS uses an unusual module system. Theories may carry parameters, and theories may include multiple instances of the same theory with different arguments, e.g., to include lists over booleans and lists over integers. All these includes use the same included identifiers, and PVS disambiguates for each occurrence of an identifiers which instance it belongs to. Consequently, internally all included identifiers carry a long list of implicit arguments that instantiate the parameters.

There is a subtle problem here: As not every identifier depends on all parameters of a theory, it is possible that identifiers included via different instances are actually equal. As that equality is undecidable, it becomes undecidable which set of identifiers has been included. Therefore, PVS's parametric includes are very difficult to eliminate in the export. Therefore, we chose to mimic them by adding a special include-declaration that abstracts over all parameters of a theory.

Finally, PVS uses primitive (co)inductive types and elaborates them into an axiomatization. That makes it possible to skip them and to export the elaborated statements only. But we still had to use some extensions of LF to represent the associated recursive functions.

8.2 System

PVS includes a very well-built export facility based on kernel-generated XML files written by Sam Owre. This is intended specifically for exports like ours as well as knowledge management applications like search. PVS was the only system encountered that already had such a system-near export facility specifically designed for that purpose by the developers.

Because PVS uses a big kernel, very little elaboration takes place. Therefore, the XML files include all user-level information enriched with the information

inferred by the kernel. This includes virtually the entire original source structure, down to source references and redundant brackets.

Because the type system in undecidable, every statement generates some type-checking-conditions, which must be proved separately. After proving these, PVS inserts them before the respective statement. Exports can choose to include or skip these additional theorems.

8.3 Libraries

PVS uses a small standard library, called the PVS prelude. Other libraries are distributed, but only the big NASA library is used widely. PVS releases are regression-tested against this library.

Our export [36] used the generated XML files for both libraries. As a part of this export collaboration, the XML schema was heavily debugged and well-documented. Concretely, we represented the PVS XML schema in Scala, and then generated MMT data structures from it directly. Our Scala implementation of the PVS XML schema is also a valuable resource as an alternative documentation of the PVS language. The export worked very smoothly (once the theoretical challenges had been resolved). A minor problem we encountered were that the source files in the NASA library must be built in a specific order, a process that PVS does not automate well yet.

PVS stores high-level proof objects in separate files. But the big prover kernel uses a lot of automation that is not recorded in these proof objects, in particular the heavy use of decision procedures. Thus, proofs have significant gaps.

8.4 Outlook and Open Challenges

PVS supports a very robust and well-documented export to XML that includes all user-level declarations. That makes it relatively easy to build and maintain exports.

The biggest challenge is that PVS uses a few **unusual language features**. This includes (co)inductive types and recursion at a similar level of difficulty as in Coq. (Coq uses dependent types and PVS uses predicate subtypes; neither subsumes the other, but both cause similar difficulties.) But it also includes an unusual kind of includes between theories, theory parameters representing theory morphisms, anonymous record types, and operators for function/record updates. While these features are quite justifiable, their subtleties make representing the language in a logical framework difficult. We were able to do so by extending the LF with corresponding features, but transporting the library to other systems remains difficult.

Just like for Mizar, a complete **export of proofs** is not possible as many proof steps are performed automatically by a large kernel and not tracked. However, once an appropriate standard as discussed in Section 2.5 exists, an export of mid-level proof terms can be realized based on XML files containing partial proof objects.

9 Conclusion

We have presented experiences from building exports of major theorem prover libraries spanning some five years. We have focused on describing the current state and open challenges in these exports with an eye towards supporting the long term community goal of integrating prover libraries and making provers interoperable.

9.1 Lessons Learned

The most important lesson to draw from our work is that prover library exports are possible but very difficult and tedious: Theoretically, they require state-of-the-art representations of logics in logical frameworks and even the design of new framework features. Here the development of Mmt, which makes implementing these logical framework feasible, was critical. Indeed the exports motivated and drove the development of new features in Mmt.

Practically, each export required enormous amounts of work both from us and from a dedicated collaborator on the system side. Each export runs into numerous subtle and/or undocumented cases that require extensive communication between these two. Here careful planning was critical to avoid running out of time, personnel, or funding half way through. Because of this high threshold to have working exports at all, many added-value services and system interoperability solutions that would otherwise be in reach (albeit still subject to substantial research efforts) can hardly be attempted at all. With our six working exports, this work can commence now, and indeed we are in contact with multiple groups that are already using them.

It is also difficult to rigorously represent the syntax and semantics of the logics of practical provers. Formal descriptions of the exact logics realized in the provers are typically incomplete, outdated, and/or distributed over multiple publications. In particular it is virtually impossible to clarify all details without talking to a system expert. We even routinely resorted to reverse-engineering the internals of the kernels to find out how corner cases are actually treated. It is even harder to then represent these logics in logical frameworks. While the latter are excellent for textbook logics such as FOL, HOL, or pure type systems, the representations of all features used in practical provers go beyond their current expressive power. This observation limits the promise of logical frameworks like LF and has been a major motivation behind the development of Mmt: it allows expanding the logical framework flexibly with the language features necessary to represent the object logic at hand, whereas any attempt to export a proof assistant library to a fixed logical framework is bound to hit expressivity limits. But even with the flexibility afforded by Mmt, the task remains very difficult. This investment is well-spent, however, as the formalization can also serve as an detailed, consolidated, and up-to-date documentation of the syntax and semantics of the logic actually used in the system.

Because of the above, it is important that exports always preserve all details of the prover's logic and refrain from mixing the export with non-compositional transformations to eliminate complex or idiosyncratic features of the logic. Such transformations may sometimes be necessary, especially to integrate libraries across

provers. But the question of how to do such transformations is in itself a difficult research question that should be separated from the engineering question of realizing any export at all.

We have not touched on logic translations and library integration in this paper, a task that may build on top of the exports we have discussed. It remains a separate and very difficult problem, and we are not sure yet which approach will prove most successful here. However, we want to stress one negative result: it does not work to give a translation from base logic L to base logic L' and then use that to automatically translate an L-library to L'. Believing that such an approach might work is a misconception we had 10 years ago ourselves and that we have encountered among many colleagues since.

For example, to translate the Coq library to PVS, it is neither sufficient nor necessary to give a translation from Coq's calculus of inductive constructions to PVS's higher-order logic. Giving such a translation is very difficult because the base logics are so different: several Coq features such as universes and dependent types cannot be expressed directly in terms of PVS features. A partial translation that leaves out those features would be both impractical and pointless as the features are used pervasively in Coq. But any translation that covers them would have to be a deep embedding, e.g., translating all Coq types to terms of a single fixed PVS type. This *would* induce a translation of the Coq library to PVS, but the image of the translation would be as semantically disconnected from the PVS library as the original Coq library was. Already the Coq propositions (which are types) would not be translated to PVS booleans, and this alignment problem only gets worse the more advanced the concepts defined in the libraries become [28]. Notably, the alignment problem exists even when there is a base logic translation such as the straightforward shallow embedding from HOL Light to PVS. Therefore, practical library translations must provide special case treatment not for the primitive features of L but for many concepts defined in the library of L. If such special treatment is done anyway, having a complete translation between the base logics becomes optional.

9.2 Future Work

Exports We are convinced that any future interoperability between proof assistants will require library exports of some kind. In the long run, the community must organize and invest efficiently the resources needed for prover developers to build and maintain such exports.

Because of resource constraints, we had to choose representative provers so far, and some like ACL2, HOL4, Lean, Metamath, and Nurpl are still prominently absent at this point. We regret being precluded from developing exports for those systems as well. Some of them would be particularly interesting, e.g., to compare the much younger Lean system to the logically similar but older Coq, or to have more libraries that are untyped (ACL2, Metamath) or have very expressive type systems (Nuprl) to compare integration problems between the different styles of typing. But we believe that our choice of systems is representative for the challenges faced during exports.

Note that logical similarly or even equivalence such as between Coq and Lean or HOL Light and HOL4 is not the most relevant indication for the challenges

by an export. While some logical issues such as the representation of the type system in the logical framework are indeed similar, the biggest challenges are often engineering aspects where even equivalent systems may differ drastically — indeed, perceived engineering improvements are often the very reason why two logically equivalent systems exists. For example, a Lean export would likely be entirely different from our Coq export whereas an export of HOL4 could be relatively similar to our HOL Light export.

In addition to what we presented here, we have so far developed additional (sometimes partial) exports for some systems including TPS, Specware, and Metamath as well as for a few computer algebra systems; while we have not discussed those here, our conclusions apply to and are informed by them as well.

Imports and Library Alignment Importing from a simple standardized representation format like ours into a prover is relatively simple in principle. However, that appearance hides the real difficulty, which is the alignment problem described in Section 9.1. Indeed, the exported libraries remain disconnected even the though they are now in the same format.

Even though the alignment problem was recognized in [50] already, which also gives the first solution in a practical setting, it has remained difficult. Major case studies so far are [61] for aligning many different small libraries and [18] for aligning two large ones. We gave a survey of issues in [28], and if anything, we see a longer list of difficulties by now.

Representation Format Our work foresees one particular integration approach, which realizes exports based on a standardized interface format, specifically OMDoc/Mmt. This is based on the assumption that the problem of library integration becomes easier to study if at least all libraries use the same format.

But further research is needed to design such intermediate formats. State of the art formats are quite strong at representing expressions (terms, types, formulas, etc.) from a diverse set of systems in a simple uniform language. But they are much weaker at representing high-level declarations and proofs. Well-designed abstractions for high-level declarations such as inductive or record types would simplify the library integration problem.

Similarly, the treatment of extra-logical declarations is difficult because systems vary so widely here that it is unclear which parts can even be reused in other libraries.

Direct Translations An alternative approach to ours is to develop direct translations without an intermediate format. Current users are already doing that ad hoc and on a small scale when they manually port developments from one system to another based on need. The community could invest into this by systematizing and automating the process.

Even though this is less attractive in principle as it would require translations for any pair of systems, this may still be sufficient in practice for two reasons. Firstly, there may never be that many major systems for which library integrations are critically needed, maybe even fewer than there are today. Secondly, even the more generic solutions via an intermediate format may end up requiring substantial library-pair-specific work anyway when fine-tuning the alignments.

In particular, we can imagine an approach similar to the automated generation by Isabelle's sledgehammer component of structured Isar proofs from automatically found proofs. Here a theorem of system L could be turned into a theorem of system L' through a combination of a formal logic translation, a database of alignments between L-symbols and L'-symbols, and some (possibly machine-learned) heuristics. The resulting theorem and proof might not be well-formed in L' and require further editing. The latter could happen manually or via an automated iteration that tries multiple variants until one of them is accepted by L'.

A logistic advantage of this approach is that it may be more appealing to the L' developer and user community for two reasons. Firstly, it has a more immediate return on investment for users. Secondly, it could plausibly yield L'-verifiable proofs faster than any approach based on an intermediate format. On the downside, due to the high resource requirements, only few systems L' might be able to sustain such translations.

9.3 Recommendations for Prover and Library Development

We have a single clear and important recommendation for developers of theorem provers and libraries: *design systems with exports in mind!*

Scalable and maintainable exports require some design decisions that are not obvious in the beginning because they often do not affect the remainder of the system. But exports typically only become interesting once a large library has been built, and at that point retrofitting export architectures to mature systems usually presents substantial difficulties. On the positive side, integrating these design recommendations into new systems is relatively easy and has overall positive side effects outside the export facility. Concretely, we recommend the following:

1. Provers should *trace user-level content* on its way through the elaboration pipeline into the kernel-level syntax. The user-level statements should remain accessible even after the corresponding kernel-level statements have been processed, and the latter should point to the former. Moreover, these cross-references should be fine-granular, ideally at the level of every single subexpression that occurs in a kernel-level statement. This would in particular ensure the availability of source references everywhere. Realizing this may however impact performance substantially and therefore must be carefully considered as a part of the overall design problem.
2. Provers should commit to *maintaining a system-near export* in some standard data format such as XML or JSON and with a well-documented schema. Importing this export should be part of regression test suites. More generally, a kernel instrumenting interface can be provided. But even then a default instrumentation should exist that builds a system-near export. The system-near export should contain all information relevant for third-party processing, ideally enough to reconstruct the original source for each fragment. If elaboration is traced as described above, it should in particular include the user-level statements.
3. Provers should *allow users to structure formalizations* with library integration in mind. Most prover logics are much more powerful than needed for most concrete formalizations. For example, the vast majority of the Coq library only uses a few universe levels, but Coq allows an arbitrary number. That

is a perfectly reasonable design for provers, but it makes it harder to reuse formalizations in other, less or differently expressive logics. But it can be difficult to retroactively check which logic features were used in a formalization, and even then it is often the case that the strength of the underlying logic has unintentionally leaked into the formalization. If a particular formalization only requires, for example, monadic second-order logic, users should be able to limit the strength of the logic in such a way that the kernel checks that the formalization stays within the intended fragment. That would allow users to control how well their formalizations, once exported, can be reused in other systems. Realizing such a feature in new provers and using it in new libraries is relatively easy, but retrofitting it to existing ones can be very difficult.

Acknowledgements Even though the colleagues with whom we worked on the exports (see the co-authors of the cited papers) were not directly involved in this paper, our discussions with them at the time have influenced this paper as well. We gratefully acknowledge project support by the German Research Council (DFG) under grants KO 2428/13-1 and RA 18723/1-1 and from the European Union under Project OpenDreamKit.

References

1. Andrews, P.B.: An Introduction to Mathematical Logic and Type Theory: to Truth through Proof. Academic Press, Orlando, Florida (1975)
2. Anonymous: The QED Manifesto. In: A. Bundy (ed.) Automated Deduction, pp. 238–251. Springer (1994)
3. Asperti, A., Guidi, F., Sacerdoti Coen, C., Tassi, E., Zacchiroli, S.: A content based mathematical search engine: Whelp. In: J.C. Filliâtre, C. Paulin-Mohring, B. Werner (eds.) Types for Proofs and Programs, International Workshop, TYPES 2004, revised selected papers, no. 3839 in LNCS, pp. 17–32. Springer Verlag (2006)
4. Asperti, A., Sacerdoti Coen, C., Tassi, E., Zacchiroli, S.: Crafting a Proof Assistant. In: T. Altenkirch, C. McBride (eds.) TYPES, pp. 18–32. Springer (2006)
5. Aspinall, D., Denney, E., Lüth, C.: A semantic basis for proof queries and transformations. In: N. Bjørner, A. Voronkov (eds.) Logic for Programming, Artificial Intelligence, and Reasoning, pp. 92–106. Springer (2012)
6. Assaf, A.: A framework for defining computational higher-order logics. Ph.D. thesis, École Polytechnique (2015)
7. Assaf, A., Burel, G.: Holide (2013). https://www.rocq.inria.fr/deducteam/Holide/index.html
8. Betzendahl, J., Kohlhase, M.: Translating the IMPS theory library to MMT/OMDoc. In: F. Rabe, W. Farmer, G. Passmore, A. Youssef (eds.) Intelligent Computer Mathematics, vol. 11006, pp. 7–22. Springer (2018)
9. Boespflug, M., Burel, G.: CoqInE: Translating the Calculus of Inductive Constructions into the lambda Pi-calculus Modulo. In: D. Pichardie, T. Weber (eds.) Proof Exchange for Theorem Proving (2012)
10. Codescu, M., Horozal, F., Kohlhase, M., Mossakowski, T., Rabe, F.: Project Abstract: Logic Atlas and Integrator (LATIN). In: J. Davenport, W. Farmer, F. Rabe, J. Urban (eds.) Intelligent Computer Mathematics, pp. 289–291. Springer (2011)
11. Condoluci, A., Kohlhase, M., Müller, D., Rabe, F., Sacerdoti Coen, C., Wenzel, M.: Relational Data Across Mathematical Libraries. In: C. Kaliszyk, E. Brady, A. Kohlhase, C. Sacerdoti Coen (eds.) Intelligent Computer Mathematics, pp. 61–76. Springer (2019)
12. Coq Development Team: The Coq Proof Assistant: Reference Manual. Tech. rep., INRIA (2015)
13. Czajka, L., Kaliszyk, C.: Hammer for Coq: Automation for dependent type theory. Journal of Automated Reasoning **61**(1-4), 423–453 (2018)
14. de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The Lean Theorem Prover (System Description). In: A. Felty, A. Middeldorp (eds.) Automated Deduction, pp. 378–388. Springer (2015)

15. Farmer, W., Guttman, J., Thayer, F.: Little Theories. In: D. Kapur (ed.) Conference on Automated Deduction, pp. 467–581 (1992)
16. Farmer, W., Guttman, J., Thayer, F.: IMPS: An Interactive Mathematical Proof System. Journal of Automated Reasoning **11**(2), 213–248 (1993)
17. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging mathematical structures. In: S. Berghofer, T. Nipkow, C. Urban, M. Wenzel (eds.) Theorem Proving in Higher Order Logics, pp. 327–342. Springer (2009)
18. Gauthier, T., Kaliszyk, C.: Aligning concepts across proof assistant libraries. Journal of Symbolic Computation **90**, 89–123 (2019)
19. Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Roux, S.L., Mahboubi, A., O'Connor, R., Biha, S.O., Pasca, I., Rideau, L., Solovyev, A., Tassi, E., Théry, L.: A Machine-Checked Proof of the Odd Order Theorem. In: S. Blazy, C. Paulin-Mohring, D. Pichardie (eds.) Interactive Theorem Proving, pp. 163–179 (2013)
20. Hales, T., Adams, M., Bauer, G., Dang, D.T., Harrison, J., Hoang, T.L., Kaliszyk, C., Magron, V., McLaughlin, S., Nguyen, T.T., Nguyen, T.Q., Nipkow, T., Obua, S., Pleso, J., Rute, J., Solovyev, A., Ta, A.T., Tran, T.N., Trieu, D.T., Urban, J., Vu, K.K., Zumkeller, R.: A formal proof of the Kepler conjecture (2014). http://arxiv.org/abs/1501.02155
21. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the Association for Computing Machinery **40**(1), 143–184 (1993)
22. Harrison, J.: HOL Light: A Tutorial Introduction. In: Proceedings of the First International Conference on Formal Methods in Computer-Aided Design, pp. 265–269. Springer (1996)
23. Horozal, F., Kohlhase, M., Rabe, F.: Extending MKM Formats at the Statement Level. In: J. Campbell, J. Carette, G. Dos Reis, J. Jeuring, P. Sojka, V. Sorge, M. Wenzel (eds.) Intelligent Computer Mathematics, pp. 64–79. Springer (2012)
24. Hurd, J.: OpenTheory: Package Management for Higher Order Logic Theories. In: G.D. Reis, L. Théry (eds.) Programming Languages for Mechanized Mathematics Systems, pp. 31–37. ACM (2009)
25. Iancu, M.: Towards Flexiformal Mathematics. Ph.D. thesis, Jacobs University Bremen (2017)
26. Iancu, M., Kohlhase, M., Rabe, F.: Translating the Mizar Mathematical Library into OMDoc format. Tech. Rep. KWARC Report-01/11, Jacobs University Bremen (2011)
27. Iancu, M., Kohlhase, M., Rabe, F., Urban, J.: The Mizar Mathematical Library in OMDoc: Translation and Applications. Journal of Automated Reasoning **50**(2), 191–202 (2013)
28. Kaliszyk, C., Kohlhase, M., Müller, D., Rabe, F.: A Standard for Aligning Mathematical Concepts. In: A. Kohlhase, M. Kohlhase, P. Libbrecht, B. Miller, F. Tompa, A. Naumowicz, W. Neuper, P. Quaresma, M. Suda (eds.) Work in Progress at CICM 2016, pp. 229–244. CEUR-WS.org (2016)
29. Kaliszyk, C., Krauss, A.: Scalable LCF-style proof translation. In: S. Blazy, C. Paulin-Mohring, D. Pichardie (eds.) Interactive Theorem Proving, pp. 51–66. Springer (2013)
30. Kaliszyk, C., Pak, K.: Semantics of Mizar as an Isabelle object logic. Journal of Automated Reasoning **63**(3), 557–595 (2019)
31. Kaliszyk, C., Rabe, F.: Towards Knowledge Management for HOL Light. In: S. Watt, J. Davenport, A. Sexton, P. Sojka, J. Urban (eds.) Intelligent Computer Mathematics, pp. 357–372. Springer (2014)
32. Kaliszyk, C., Urban, J.: HOL(y)hammer: Online ATP service for HOL light. Mathematics in Computer Science **9**(1), 5–22 (2015)
33. Keller, C., Werner, B.: Importing HOL Light into Coq. In: M. Kaufmann, L. Paulson (eds.) Interactive Theorem Proving, pp. 307–322. Springer (2010)
34. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an operating-system kernel. Communications of the ACM **53**(6), 107–115 (2010)
35. Kohlhase, M.: OMDoc: An Open Markup Format for Mathematical Documents (Version 1.2). No. 4180 in Lecture Notes in Artificial Intelligence. Springer (2006)
36. Kohlhase, M., Müller, D., Owre, S., Rabe, F.: Making PVS Accessible to Generic Services by Interpretation in a Universal Format. In: M. Ayala-Rincon, C. Munoz (eds.) Interactive Theorem Proving, pp. 319–335. Springer (2017)
37. Kohlhase, M., Rabe, F.: QED Reloaded: Towards a Pluralistic Formal Library of Mathematical Knowledge. Journal of Formalized Reasoning **9**(1), 201–234 (2016)

38. Kohlhase, M., Rabe, F., Wenzel, M.: Making Isabelle content accessible in knowledge representation formats. In: M. Bezem, A. Mahboubi (eds.) Proceedings of the 25th International Conference on Types for Proofs and Programs, TYPES 2019, *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 175. Dagstuhl Publishing (2020). DOI 10.4230/LIPIcs.TYPES.2019.1. URL https://drops.dagstuhl.de/opus/volltexte/2020/13065
39. Krauss, A., Schropp, A.: A Mechanized Translation from Higher-Order Logic to Set Theory. In: M. Kaufmann, L. Paulson (eds.) Interactive Theorem Proving, pp. 323–338. Springer (2010)
40. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7), 107–115 (2009)
41. Lewis, R.: An extensible ad hoc interface between Lean and Mathematica. In: C. Dubois, B.W. Paleo (eds.) Proof eXchange for Theorem Proving, pp. 23–37. Electronic Proceedings in Theoretical Computer Science (2017)
42. Li, Y.: IMPS to OMDoc translation (2002). Bachelor's Thesis, McMaster University
43. Müller, D.: Mathematical knowledge management across formal libraries. Ph.D. thesis, Informatics, FAU Erlangen-Nürnberg (2019). URL https://opus4.kobv.de/opus4-fau/files/12359/thesis.pdf
44. Müller, D., Rabe, F., Kohlhase, M.: Theories as types. In: D. Galmiche, S. Schulz, R. Sebastiani (eds.) 9th International Joint Conference on Automated Reasoning. Springer Verlag (2018). URL https://kwarc.info/kohlhase/papers/ijcar18-records.pdf
45. Müller, D., Rabe, F., Rothgang, C., Kohlhase, M.: Representing structural language features in formal meta-languages. In: C. Benzmüller, B. Miller (eds.) Intelligent Computer Mathematics (CICM) 2020, *LNAI*, vol. 12236, pp. 206–221. Springer (2020). URL https://kwarc.info/kohlhase/papers/cicm20-features.pdf
46. Müller, D., Rabe, F., Sacerdoti Coen, C.: The Coq Library as a Theory Graph. In: C. Kaliszyk, E. Brady, A. Kohlhase, C. Sacerdoti Coen (eds.) Intelligent Computer Mathematics, pp. 171–186. Springer (2019)
47. Naumov, P., Stehr, M., Meseguer, J.: The HOL/NuPRL proof translator - a practical approach to formal interoperability. In: R. Boulton, P. Jackson (eds.) 14th International Conference on Theorem Proving in Higher Order Logics. Springer (2001)
48. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Springer (2002)
49. Obua, S., Adams, M., Aspinall, D.: Capturing hiproofs in HOL Light. In: MKM/Calculemus/DML, pp. 184–199 (2013)
50. Obua, S., Skalberg, S.: Importing HOL into Isabelle/HOL. In: N. Shankar, U. Furbach (eds.) Automated Reasoning, vol. 4130. Springer (2006)
51. Owre, S., Rushby, J., Shankar, N.: PVS: A Prototype Verification System. In: D. Kapur (ed.) 11th International Conference on Automated Deduction (CADE), pp. 748–752. Springer (1992)
52. Paulson, L.: Isabelle: The Next 700 Theorem Provers. In: P. Odifreddi (ed.) Logic and Computer Science, pp. 361–386. Academic Press (1990)
53. Paulson, L.: Isabelle: A Generic Theorem Prover, *Lecture Notes in Computer Science*, vol. 828. Springer (1994)
54. Paulson, L., Coen, M.: Zermelo-Fraenkel Set Theory (1993). Isabelle distribution, ZF/ZF.thy
55. Rabe, F.: The MMT API: A Generic MKM System. In: J. Carette, D. Aspinall, C. Lange, P. Sojka, W. Windsteiger (eds.) Intelligent Computer Mathematics, pp. 339–343. Springer (2013)
56. Rabe, F., Kohlhase, M.: A Scalable Module System. Information and Computation **230**(1), 1–54 (2013)
57. Rahli, V., Cohen, L., Bickford, M.: A verified theorem prover backend supported by a monotonic library. In: G. Barthe, G. Sutcliffe, M. Veanes (eds.) Logic for Programming, Artificial Intelligence and Reasoning, pp. 564–582. EasyChair (2018)
58. RDF Core Working Group of the W3C: Resource Description Framework Specification (2004). http://www.w3.org/RDF/
59. Sacerdoti Coen, C.: A Plugin to Export Coq Libraries to XML. In: K. C, E. Brady, A. Kohlhase, C. Sacerdoti Coen (eds.) Intelligent Computer Mathematics, pp. 243–257. Springer (2019)
60. Schürmann, C., Stehr, M.: An Executable Formalization of the HOL/Nuprl Connection in the Metalogical Framework Twelf. In: F. Baader, A. Voronkov (eds.) 11th International Conference on Logic for Programming Artificial Intelligence and Reasoning. Springer (2004)

61. Thiré, F.: Sharing a library between proof assistants: Reaching out to the HOL family. In: F. Blanqui, G. Reis (eds.) Logical Frameworks and Meta-Languages: Theory and Practice, pp. 57–71. EPTCS (2018)
62. Trybulec, A., Blair, H.: Computer Assisted Reasoning with MIZAR. In: A. Joshi (ed.) Proceedings of the 9th International Joint Conference on Artificial Intelligence, pp. 26–28. Morgan Kaufmann (1985)
63. Urban, J.: Translating Mizar for First Order Theorem Provers. In: A. Asperti, B. Buchberger, J. Davenport (eds.) Mathematical Knowledge Management, pp. 203–215. Springer (2003)
64. W3C: SPARQL Query Language for RDF (2008). http://www.w3.org/TR/rdf-sparql-query/