Boosting the Reuse of Formal Specifications

Mariano M. Moscato[1(✉)], Carlos G. Lopez Pombo[2], César A. Muñoz[3], and Marco A. Feliú[1]

[1] National Institute of Aerospace, Hampton, VA, USA
{mariano.moscato,marco.feliu}@nianet.org[⋆]

[2] CONICET–Universidad de Buenos Aires, Instituto de Investigación en Ciencias de la Computación (ICC), Buenos Aires, Argentina

[3] NASA Langley Research Center, Hampton, VA, USA
cesar.a.munoz@nasa.gov

Abstract. Advances in theorem proving have enabled the emergence of a variety of formal developments that, over the years, have resulted in large corpuses of formalizations. For example, the NASA PVS Library is a collection of 55 formal developments written in the Prototype Verification System (PVS) over a period of almost 30 years and containing more than 28000 proofs. Unfortunately, the simple accumulation of formal developments does not guarantee their reusability. In fact, in formal systems with very expressive specification languages, it is often the case that a particular conceptual object is defined in different ways. This paper presents a technique to establish sound connections between formal definitions. Such connections support the possibility of (partial) borrowing of proved results from one formal description into another, improving the reusability of formal developments. The technique is described using concepts from the field of universal algebra and algebraic specification. The technique is illustrated with concrete examples taken from formalizations available in the NASA PVS Library.

1 Introduction

Proof assistants have been actively used for decades now. Advances in formal verification techniques have enabled their use in the development cycle of critical systems. The routine use of proof assistants in some domains has resulted in the generation of a large number of formalizations. This generation of content can be seen as an unguided collective effort, since it includes the work of very different actors, from purely academic environments to industrial organizations. Thus, each formalization is biased by the particular background, goals, and style of its creators and the subtleties of each theorem prover.

With the aim of promoting the reuse of existing efforts, large corpuses of formal models have been created and augmented by accumulating individual endeavors. For instance, the NASA PVS Library[4] is a collection of formal models

[⋆] Research by the first and fourth authors was supported by the National Aeronautics and Space Administration under NASA/NIA Cooperative Agreement NNL09AA00A.

[4] https://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/

written in the Prototype Verification System (PVS) [1] and maintained by the NASA Langley Formal Methods Team. While this library is extensively used at NASA and other places, it falls short in reusability. This is often the case of formal systems featuring powerful formalisms such as higher-order logic. In such settings, the same conceptual object can be described in different and, sometimes, incompatible ways. While some of these differences can be avoided, they are often intentional. For example, a particular way of stating some definition could help the construction of a proof for a specific property, but could make the definition not suitable for computation. Examples of this phenomenon arise naturally when working with structured data types. Depending on the objectives of a particular project, it may be more convenient to represent a graph with a set of nodes and a set of edges, while in a different context, it may be preferable to use an adjacency list instead. Several examples of multiple definitions for the same conceptual element can be found in the NASA PVS Library.

This paper proposes a technique for *(1)* stating a formal connection between (parts of) different specifications, which provide the same functionality, and *(2)* supporting the transference of properties (and their proofs) between these specifications. From a practical point of view, the proposed technique improves the possibility of reuse of existing developments. This technique has the following distinguishing features: it is *explicitly verifiable*, since it provides a formal proof of the correctness of the link between definitions that can be checked in the same environment (PVS), it is *nonintrusive*, since its use does not require any modification of existing developments, it is *automatable*, since most of the steps of the proposed technique are suitable for automation, and it is *general* enough to deal with cases that could not be adressed by similar techniques.

The rest of the paper is devoted to the description of the representation technique and the illustration of its use by means of a practical case study. In Section 2, basic definitions are presented in order to state the notation to be used in the following sections. Section 3 describes formally the datatype connection technique. A real case study with significative practical consequences is detailed in Section 4. Section 5 discusses related work. Finally, Section 6 summarizes the results and discusses further work.

2 Preliminary Definitions

Most of the definitions and results presented in this section are taken and adapted from [2, 3]. The formalism used throughout the rest of the paper is based on higher-order logic and its syntax and semantics follows mainly [4].

A concrete PVS specification is used to illustrate the notions and concepts introduced in this and the next section. PVS supports a strongly-typed higher-order language with several additional features intended to provide the user with a rich set of tools for the formalization of concepts. Regarding datatype definitions, PVS includes built-in support for structured, algebraic, and enumeration types, among other characteristics. As in the vast majority of modern computerized deduction systems, specifications in PVS must be grouped in syntactic

```
fseqs [T: TYPE+]: THEORY BEGIN
  [...]
  default: T = choose({t:T | TRUE})

  barray(n: nat): TYPE = {f: [nat->T] |
   FORALL (i: nat): i >= n => f(i) = default}

  fseq:TYPE = [#length:nat, seq:barray(length)#]

  empty?_fseq(f): bool = (f‘length = 0)
  [...]
END fseqs
```

```
list [T: TYPE]: DATATYPE BEGIN
  null: null?
  cons (car: T, cdr:list):cons?
END list
```

Fig. 1. Fragments of different PVS theories representing finite-sized containers. Left: finite sequences (NASA PVS Library). Right: finite lists (PVS Prelude).

constructions called *theories.* A comprehensive battery of theories containing fundamental definitions and properties is provided under the name of *PVS prelude.* There, notions for basic datatypes such as booleans, numbers, and characters, and well-known data structures, such as arrays and lists, can be found. All the definitions in the prelude are implicitly available to any user-defined theory. Additionally, the NASA PVS Library is an ongoing effort that collects a significant amount of both elemental and specialized PVS developments. In order to use definitions from user-defined theories, they must be explicitly imported. To improve the presentation of the notions needed to describe the representation technique, practical aspects such as the structured nature of PVS specifications are not reflected in the theoretical development. Nevertheless, comments on how to bridge that gap are provided when it is considered adequate.

Figure 1 presents the running example for this and the next section[5]. On its left-hand side, an excerpt from a formalization of finite sequences by Butler and Maddalon is depicted. This theory, which is part of the NASA PVS Library, takes as parameter the type (`T`) of the elements being contained in the sequence. The definition of the type of the finite sequences (`fseq`) is based on an auxiliary type named `barray`, for *bounded array.* The arrays in PVS are formalized as functions from natural numbers to `T`. The `barray` datatype depends on a natural number, which represents the bound of the array; queries for an index beyond the bound are defined to result in an arbitrary, but constant, value (`default`). The existence of such value is guaranteed by the declaration of `T` as `TYPE+`, which forces `T` to be a nonempty type. The right side of the figure shows the formalization of an algebraic datatype representing finite lists of a generic type `T`. In this case, the `DATATYPE` construct is used as an abbreviation for the definition of a regular PVS theory, containing a type (`list`), a constant (`null`), three functions (`cons`, `car`, and `cdr`) and two predicates (`null?` and `cons?`). The constant `null` denotes the empty list and the function `cons` can be used to construct a list from an element and another list. The predicates `null?` and `cons?` determine whether a given

[5] Keywords in PVS are not case sensitive. Uppercase is used here to differentiate them from the rest of the tokens.

list is empty or not, respectively. The function `car` returns the first element of a list. The function `cdr` returns the rest of its elements.

Definition 1 (Type). *Let T be a set of sort symbols. $Type(T)$ is defined as the smallest set satisfying: (1) $T \subseteq Type(T)$, and (2) if $t_1, \cdots, t_k \in Type(T)$, then the list $[t_1, \cdots, t_k] \in Type(T)$.*

Definition 2 (Signature of Symbols). *Let T be a set of sort symbols and $s, t \in Type(T)$. The* signature *(or type) of a function symbol f is of the form $[s, t]$ and is also written as $[s \to t]$, while the* signature *(or type) of a predicate symbol p is of the form $[t]$.*

Definition 3 (Signature of a Language). *Let T be a set of sort symbols. The* signature *of a language is a structure $\langle \mathcal{C}, \mathcal{F}, \mathcal{P} \rangle$ where $\mathcal{C}$ is a set of constant symbols and $\mathcal{F}$ (resp. $\mathcal{P}$) is an indexed set of function (resp. predicate) symbols, each one with its corresponding type over $Type(T)$ accessible through the function $type : \mathcal{C} \cup \mathcal{F} \cup \mathcal{P} \to Type(T)$.*

Henceforth, whenever a signature Π is used, its sets of symbols will be referred to as $\mathcal{C}_\Pi$, $\mathcal{F}_\Pi$ and $\mathcal{P}_\Pi$. The notion of language signature refers to the available symbols in the context of a given PVS theory. Thus, for example, the signature $\Pi_{\texttt{fseqs}}$ includes all the symbols defined in the PVS prelude, the constant symbol `default`, and the predicate symbol `empty?_fseq`. The set $T_{\texttt{fseqs}}$ of sort symbols, on which the signature $\Pi_{\texttt{fseqs}}$ stands, contains all the sort symbols defined in the prelude plus `barray` and `fseq`. Because `list` is part of the PVS prelude, its corresponding set of sort symbols $T_{\texttt{list}}$ and signature $\Pi_{\texttt{list}}$ contain the sort and language symbols previously defined, plus the sort symbol `list` and the constant, function, and predicate symbols mentioned above. Thus, $\Pi_{\texttt{list}} \subset \Pi_{\texttt{fseqs}}$.

Definition 4 (The Language of Higher-Order Logic with Equality). *Let T be a set of sort symbols and Π a language signature over $Type(T)$. Let $\mathcal{X}$ be a set of flexible symbols for which the function $type : \mathcal{X} \to Type(T)$ reports the type of each symbol. $Term(\Pi, \mathcal{X})$ is defined as the smallest set satisfying:*

- *$\mathcal{X} \subseteq Term(\Pi, \mathcal{X})$, $\mathcal{C}_\Pi \subseteq Term(\Pi, \mathcal{X})$, and $\mathcal{F}_\Pi \subseteq Term(\Pi, \mathcal{X})$,*
- *for any function symbol f in $\mathcal{F}_\Pi$ s.t. $type(f) = [t \to t_{k+1}]$ and $t = [t_1, \cdots, t_k]$, for all $r_i \in Term(\Pi, \mathcal{X})$ with $1 \leq i < k$ s.t. $type(r_i) = t_i$, $f(r_1, \cdots, r_k) \in Term(\Pi, \mathcal{X})$ and $type(f(r_1, \cdots, r_k)) = t_{k+1}$, and*
- *for any flexible symbol x in $\mathcal{X}$ s.t. $type(x) = [t \to t_{k+1}]$ and $t = [t_1, \cdots, t_k]$, for all $r_i \in Term(\Pi, \mathcal{X})$ with $1 \leq i < k$ s.t. $type(r_i) = t_i$, $x(r_1, \cdots, r_k) \in Term(\Pi, \mathcal{X})$ and $type(x(r_1, \cdots, r_k)) = t_{k+1}$.*

In its turn, $Form(\Pi, \mathcal{X})$ is the smallest set satisfying:

- *for any type t in $Type(T)$, $\{r, r'\} \subseteq Term(\Pi, \mathcal{X})$, and $type(r) = type(r') = t$, $r = r' \in Form(\Pi, \mathcal{X})$, and*
- *for any predicate symbol p in $\mathcal{P}_\Pi$ s.t. $type(p) = [t_1, \cdots, t_k]$, for all $r_i \in Term(\Pi, \mathcal{X})$ with $1 \leq i < k$ s.t. $type(r_i) = t_i$, $p(r_1, \cdots, r_k) \in Form(\Pi, \mathcal{X})$*

– *for any formulas φ and ψ in $\mathit{Form}(\Pi, \mathcal{X})$, $t \in \mathit{Type}(T)$ and $x \in \mathcal{X}$, $\{\neg\varphi, \varphi \vee \psi, (\exists x : t)\varphi\} \subseteq \mathit{Form}(\Pi, \mathcal{X})$.*

The language of higher-order logic over signature Π and a set of flexible symbols $\mathcal{X}$ will be denoted $\mathcal{L}_{\Pi,\mathcal{X}}$.

Due to space limitations, and because it is not central to the proposed work, a formal definition of model theory for the language of higher-order logic of Definition 4 is not explicitly presented. The next definition presents the classical proof calculus for higher-order logics which mainly follows [5, Chap. 11].

Definition 5 (Proof Theory). *Let T be a set of sort symbols and Π be a language signature over $\mathit{Type}(T)$. The notion of* syntactic deduction *is expressed by the relation $\vdash \subseteq 2^{\mathit{Form}(\Pi,\mathcal{X})} \times \mathit{Form}(\Pi, \mathcal{X})$ and defined by the following set of rules: (1) the usual rules for first-order logic [5, Sec. 2.1.1] and (2) rules for the introduction and elimination of higher-order quantifiers: see [5, Sec. 11.1.1] for a description of the rules and its interpretation in terms of substitution using lambda expressions and their application [6, Sec. 3.1] or [7, Chap. 4].*

The calculus implemented in PVS is a version of sequent calculus for higher-order logic, which fulfill the requisites stated by the Definition 5.

Definition 6 (Theory Presentation). *Let T a set of sort symbols and Π a language signature over $\mathit{Type}(T)$. A* theory presentation *over flexible symbols $\mathcal{X}$ is a structure $\langle T, \Pi, \Gamma, \Delta\rangle$, where $\Gamma \cup \Delta \subseteq \mathcal{L}_{\Pi,\mathcal{X}}$ and $\{\Gamma \vdash \varphi\}_{\varphi\in\Delta}$.*

A theory presentation in the context of PVS involves all the symbols defined in a given theory as well as all the symbols available from other (imported) theories, along with all the explicit and implicit axioms, introduced for example by defining types, constants, predicates, or functions, and the properties stated as theorems.

3 Representation Technique

Both types from the example of Figure 1, `list` and `fseq`, can be seen as formalizations of the same ideal object: an ordered finite-sized container. Nevertheless, they are fairly different from a practical point of view. PVS supports the evaluation of ground expressions through a translation to Common Lisp, which enables useful features such as rapid prototyping and computational reflection [8]. Expressions involving `fseq` cannot be evaluated since they are based on total functions over the infinite domain of natural numbers. On the other hand, expressions involving `list` are completely amenable to evaluation because of its recursive definition. However, proving properties of `list` often requires induction, while the same properties in `fseq` are proven by straightforward instantiations of existential or universal quantifications. Hence, it is useful to have a connection between `fseq` and `list` in order to transfer the functions and properties defined on `fseq` to `list` with minimum human interaction.

The representation of theory presentations, as stated in this paper, finds its theoretical foundations in the field of algebraic specification [9, 10] and more specifically in its subsequent development through the use of category theory [11, 12]. The definitions shown below constitute the basic elements used in constructing representations between theory presentations.

Definition 7 (Type Map). *Let T and T' be sets of sort symbols. The function $\tau : T \to \mathit{Type}(T')$ is a* type map *if it is a total function mapping sort symbols in T to types in $\mathit{Type}(T')$. Given a type map τ, its homomorphic extension to lists of types will be denoted by $\widehat{\tau_T^{T'}} : \mathit{Type}(T) \to \mathit{Type}(T')$.*

Continuing with the example of Figure 1, the following equations should hold: *1)* $\widehat{\tau_{\mathtt{fseq}}^{\mathtt{list}}}(\mathtt{fseq}) = \mathtt{list}$ and *2)* $\widehat{\tau_{\mathtt{fseq}}^{\mathtt{list}}}(t) = t$ for any other $t \in T_{\mathtt{fseqs}}$. The extension of $\widehat{\tau_{\mathtt{fseq}}^{\mathtt{list}}}$ to lists of types is done positionwise.

Definition 8 (Language Signature Map). *Let T and T' be sets of sort symbols and $\Pi = \langle \mathcal{C}, \mathcal{F}, \mathcal{P} \rangle$ and $\Pi' = \langle \mathcal{C}', \mathcal{F}', \mathcal{P}' \rangle$ be language signatures over $\mathit{Type}(T)$ and $\mathit{Type}(T')$ respectively. Let $\mathcal{X}$ and $\mathcal{X}'$ be sets of flexible symbols and $\tau : T \to \mathit{Type}(T')$ be a type map.*

A total function $\left\langle \sigma^{\mathcal{C}}, \sigma^{\mathcal{F}}, \sigma^{\mathcal{P}} \right\rangle_\tau : \Pi \to \mathit{Term}(\Pi', \mathcal{X}') \cup \mathit{Form}(\Pi', \mathcal{X}')$ is a language signature map *if it satisfies: 1) $\sigma^{\mathcal{C}} : \mathcal{C} \to \mathit{Term}(\Pi', \mathcal{X}')$ is a total function s.t. $\widehat{\tau_T^{T'}}(\mathit{type}(c)) = \mathit{type}(r')$, whenever $\sigma^C(c) = r'$, 2) $\sigma^{\mathcal{F}} : \mathcal{F} \to \mathit{Term}(\Pi', \mathcal{X}')$ is a total function s.t. $\widehat{\tau_T^{T'}}(\mathit{type}(f)) = \mathit{type}(r')$, whenever $\sigma^F(f) = r'$, and 3) $\sigma^{\mathcal{P}} : \mathcal{P} \to \mathit{Form}(\Pi', \mathcal{X}')$ is a total function s.t. $\widehat{\tau_T^{T'}}(\mathit{type}(p)) = [t'_1, \ldots, t'_k]$, whenever $\sigma^P(p) = \varphi'$, $\mathit{fv}(\varphi') = \{x'_1, \ldots, x'_k\}$, where $\mathit{fv} : \mathit{Form}(\Pi', \mathcal{X}') \to \mathcal{X}'$ is a function that yields the free variables of a formula and $\mathit{type}(x'_i) = t'_i$.*

Language signature maps show how constants, functions and predicates from the source theory presentation are interpreted in the target one. As the target theory presentation is not guaranteed to have the exact same signature, constant and function symbols from the source theory can be represented by terms in the target. The same consideration applies to predicate symbols from the source theory. In the example, the only predicate defined in the source theory (`fseqs`) is `empty?_fseq` and holds when the sequence is empty. This symbol is represented by the term $\mathtt{null?}(l)$, whose free variable l is of type `list`, as stated by the type map $\widehat{\tau_{\mathtt{fseq}}^{\mathtt{list}}}$. The notion of term representation, defined below, relates a term in the source language with its intended representation in the target language.

Definition 9 (Term Representation). *Let T and T' be two sets of sort symbols and $\Pi = \langle \mathcal{C}, \mathcal{F}, \mathcal{P} \rangle$ and $\Pi' = \langle \mathcal{C}', \mathcal{F}', \mathcal{P}' \rangle$ be language signatures over $\mathit{Type}(T)$ and $\mathit{Type}(T')$ respectively. Let $\tau : T \to \mathit{Type}(T')$ be a type map, $\sigma = \left\langle \sigma^{\mathcal{C}}, \sigma^{\mathcal{F}}, \sigma^{\mathcal{P}} \right\rangle_\tau$ be a language signature map, and $\mathcal{X}$ and $\mathcal{X}'$ be sets of flexible symbols. The* term representation relation *$\mathit{Repr}_\sigma \subseteq \mathit{Term}(\Pi, \mathcal{X}) \times \mathit{Term}(\Pi', \mathcal{X}')$ is the smallest relation such that:*

(1) it relates flexible symbols in $\mathcal{X}$ to flexible symbols in $\mathcal{X}'$ preserving:
   (a) typing, i.e., for all $x \in \mathcal{X}$ and $x' \in \mathcal{X}'$, if $\mathit{Repr}_\sigma(x, x')$ then $\widehat{\tau_T^{T'}}(\mathit{type}(x)) = \mathit{type}(x')$, and

(b) free occurrences of symbols, i.e., if $Repr_\sigma(t,t')$ each free flexible symbol in t is related to the same flexible symbol from $\mathcal{X}'$ via $Repr_\sigma$,

(2) it is homomorphic with respect to language signature representation, i.e.,

- *for all function symbol f in $\mathcal{F}$ such that $type(f) = [t_1, \cdots, t_k \to t_{k+1}]$, and for all $r_i \in Term(\Pi, \mathcal{X})$ and $r'_i \in Term(\Pi', \mathcal{X}')$ s.t. $type(r_i) = t_i$, $type(r'_i) = t'_i$, and $Repr_\sigma(r_i, r'_i)$ holds for each $i \in \mathbb{N}$ such that $1 \leq i \leq k$,*

$$Repr_\sigma(f(r_1, \cdots, r_k), (\lambda x'_1 \cdots x'_k.\sigma^{\mathcal{F}}(f))(r'_1, \cdots, r'_k)) \text{ holds,}$$

  being $x'_1, \cdots, x'_{k-1}$ the free flexible symbols occurring in $\sigma^{\mathcal{F}}(f)$.

- *for all predicate symbol p in $\mathcal{P}$ such that $type(p) = [t_1, \cdots, t_k]$, and for all $r_i \in Term(\Pi, \mathcal{X})$ and $r'_i \in Term(\Pi', \mathcal{X}')$ s.t. $type(r_i) = t_i$, $type(r'_i) = t'_i$, and $Repr_\sigma(r_i, r'_i)$ holds for each $i \in \mathbb{N}$ such that $1 \leq i \leq k$,*

$$p(r_1, \cdots, r_k) \textit{ iff } \sigma^{\mathcal{P}}(p)\Big|^{[r'_1, \cdots, r'_k]}_{[x'_1, \cdots, x'_k]}$$

  being $x'_1, \cdots, x'_k$ the free flexible symbols occurring in $\sigma^{\mathcal{P}}(p)$.

A term representation relation $Repr_\sigma$ relies on the signature map σ on which it is constructed. In fact, $Repr_\sigma$ inherits several of the properties fulfilled by σ. For example, since signature maps are type-preserving (see Def. 8) and condition (1) above assures type-preservation in the case of representation of flexible symbols, it can be assured that $Repr_\sigma$ preserves typing as well, i.e., if $Repr_\sigma(t, t')$ then $\widehat{\tau_T^{T'}}(t) = t'$. Additionally, properties such as totality, injectivity and surjectivity can be inherited from signature maps to term representation relations. In the following, the case of surjective term-representation relations is addressed first in order to ease the reading. A more general case is explained later.

The representation of formulas is just a translation between two different theory presentations but within the same logical language. The only nontrivial cases require the application of language signature maps, for the case of predicate symbols, and term representation as it was defined above.

Definition 10 (Formula Representation). *Let T and T' be sets of sort symbols, $\Pi = \langle \mathcal{C}, \mathcal{F}, \mathcal{P} \rangle$ and $\Pi' = \langle \mathcal{C}', \mathcal{F}', \mathcal{P}' \rangle$ be language signatures over $Type(T)$ and $Type(T')$ respectively, $\tau : T \to Type(T')$ be a type map, $\sigma = \left\langle \sigma^{\mathcal{C}}, \sigma^{\mathcal{F}}, \sigma^{\mathcal{P}} \right\rangle_\tau$ be a surjective language signature map, and $\mathcal{X}$ and $\mathcal{X}'$ be sets of flexible symbols. A* formula representation *$Tr_\sigma : \mathcal{L}(\Pi, \mathcal{X}) \to \mathcal{L}(\Pi', \mathcal{X}')$ is defined as follows:*

$$\begin{aligned}
&Tr_\sigma(r_1 = r_2) = r'_1 = r'_2 && \text{, s.t. } Repr_\sigma(r_i, r'_i) \text{ for } i \in \{1,2\}.\\
&Tr_\sigma(p(r_1, \cdots, r_k)) = \sigma^{\mathcal{P}}(p)\Big|^{[r'_1, \cdots, r'_k]}_{[x'_1, \cdots, x'_k]} && \text{, for all } p \in \mathcal{P} \text{ and}\\
& && \quad Repr_\sigma(r_i, r'_i) \text{ for } i \in \{1, \cdots, k\}.\\
&Tr_\sigma(\varphi \vee \psi) = Tr_\sigma(\varphi) \vee Tr_\sigma(\psi) &&\\
&Tr_\sigma((\exists x : t)\varphi) = (\exists\ x' : \widehat{\tau_T^{T'}}(t))\, Tr_\sigma(\varphi), \text{ for all } x \in \mathcal{X}. &&
\end{aligned}$$

Tr_σ extends to sets of formulas as $Tr_\sigma(\Delta) = \{ Tr_\sigma(\varphi) \mid \varphi \in \Delta\}$.

The next definition provides the means for connecting two theory presentations. This type of map was introduced in [13] under the name *map of entailment system*. More recently, a similar type of map used to connect the semantics of two theory presentations, was given the name of *theoroidal co-morphisms of institutions* by Goguen and Roçu [14].

Definition 11 (Theoroidal Representation). *Let T and T' be sets of sort symbols, Π and Π' be language signatures over $Type(T)$ and $Type(T')$ respectively, $\tau : T \to Type(T')$ be a type map, σ be a language signature map, and $\mathcal{T} = \langle T, \Pi, \Gamma, \Delta\rangle$ and $\mathcal{T}' = \langle T', \Pi', \Gamma', \Delta'\rangle$ two theory presentations. Then, a formula representation Tr_σ is said to be a* theoroidal representation *if $\Gamma' \vdash Tr_\sigma(\Gamma)$. It is said to be* axiom preserving *if $Tr_\sigma(\Gamma) \subseteq \Gamma'$.*

Note that if σ is a surjective signature map, Tr_σ is axiom preserving. It can be shown that theoroidal representations compose in a very smooth component-wise form yielding theoroidal representations (see [13, pp. 24] for details). Theorem 1 states that theoroidal representations preserve deductibility, i.e., the existence of a proof, while axiom preserving theoroidal representations are stronger by providing proof representation.

Theorem 1 (Deductibility Preservation). *Let T and T' be sets of sort symbols, Π and Π' be language signatures over $Type(T)$ and $Type(T')$ respectively, $\tau : T \to Type(T')$ be a type map, σ be a language signature map, and $\mathcal{T} = \langle T, \Pi, \Gamma, \Delta\rangle$ and $\mathcal{T}' = \langle T', \Pi', \Gamma', \Delta'\rangle$ two theory presentations related by the theoroidal representation Tr_σ. Then,*

$$\Gamma \vdash \varphi \quad \text{implies} \quad \Gamma' \vdash Tr_\sigma(\varphi).$$

Proof. First observe that $\Gamma' \vdash Tr_\sigma(\Gamma)$ holds by Def. 11 and then, $Tr_\sigma(\Gamma) \vdash Tr_\sigma(\varphi)$ follows by induction on the structure of the proof. The base case is when $\varphi \in \Gamma$, which holds by definition of $Tr_\sigma(\Gamma)$ in Def. 10. The inductive steps follow by considering each of the rules for introducing and eliminating logical symbols (i.e., $\neg$, $\vee$ and $\exists$). The inductive hypothesis guarantees that the hypothesis of the rule follow from $Tr_\sigma(\Gamma)$. After applying the same rule, and considering that Tr_σ preserve the logical structure of the formula, its definition can be fold to obtain the translation of the original formula. □

Corollary 1 (Theorem Preservation). *Let T and T' be sets of sort symbols and Π and Π' be language signatures over $Type(T)$ and $Type(T')$ respectively, $\tau : T \to Type(T')$ be a type map, σ be a language signature map, and $\mathcal{T} = \langle T, \Pi, \Gamma, \Delta\rangle$ and $\mathcal{T}' = \langle T', \Pi', \Gamma', \Delta'\rangle$ two theory presentations related by the axiom preserving theoroidal representation Tr_σ. Then,*

$$\varphi \in \Delta \quad \text{implies} \quad Tr_\sigma(\varphi) \in \Delta'.$$

Proof. The proof follows trivially from Thm. 1 and by definition of axiom preserving theoroidal representations in Def. 11. □

A mechanizable method for transferring theorems between theory presentations can be inferred from the structure of the proof of Theorem 1. The mechanization is particularly easier to achieve when dealing with conditions compatible with the hypothesis of Corollary 1.

Up to this point, some strong restrictions were posed on mapping notions presented in this section, i.e., type maps, language signature maps, term representations, formula representations, and theoroidal representations, in order to ease the presentation of the technique. These restrictions are totality and surjectivity. Establishing a theorem preserving representation from one theory to another should be more flexible.

Relaxing the restriction about surjectivity of representations implies that no every element in the target domain is required to represent an element in the source domain. To allow such relaxation, the way in which formulas are translated (Def. 10) needs to be modified. In particular, quantified variables appearing in the representation of a quantified formula must range only over those elements from $\widehat{\tau_T^{T'}}(t)$ that in fact represent elements from t. Then, a way to refer to representability of elements must be included at the logical level of the language of the signatures. While in higher-order settings such as the one provided by PVS is possible to define the notions needed at that level (symbol, term, formula, term representation, etc.), an alternative way is proposed in order to reduce the amount of practical effort needed to apply the technique. This alternative approach is based on the semantic counterpart of the *Repr* relation. Given a source and a target theory representation, Π and Π' respectively, the semantic version of the term representation relation (*Repr*) can be stated as a relation $\overline{repr} \subseteq \overline{Type(T)} \times \overline{Type(T')}$ such that:

(1) for every pair of predicates $\overline{p} \in \{\overline{p}\}_{p \in \mathcal{P}}$ and $\overline{p'} \in \{\overline{p'}\}_{p' \in \mathcal{P}'}$ s.t. $\sigma^{\mathcal{P}}(p) = p'$, and for all $x_i \in \overline{Type(T)}$, with $1 \leq i \leq n$ and $n = arity(p)$, and $x'_i \in \overline{Type(T')}$ s.t. $\overline{repr}(x_i, x'_i)$,

$$\overline{p}(x_1, \cdots, x_n) \textit{ iff } \overline{p'}(x'_1, \cdots, x'_n) \qquad \text{and}$$

(2) for every pair of functions $\overline{f} \in \{\overline{f}\}_{f \in \mathcal{F}}$ and $\overline{f'} \in \{\overline{f'}\}_{f' \in \mathcal{F}'}$ s.t. $\sigma^{\mathcal{F}}(f) = f'$, and for all $x_i \in \overline{Type(T)}$ and $x'_i \in \overline{Type(T')}$ s.t. $\overline{repr}(x_i, x'_i)$ with $1 \leq i \leq n$ and $n = arity(f)$,

$$\overline{repr}(\overline{f}(x_1, \cdots, x_n), \overline{f'}(x'_1, \cdots, x'_n)).$$

The similarity between the conditions on $\overline{repr}$ and *Repr* is not casual and can be used to prove its equivalence. The PVS definition shown in the left side of Fig. 2 can be proposed to denote $\overline{repr}$ for the example. The lemma depicted on the right side is to be proved to assure that it is in fact a good candidate. Then, the representation of a quantified formula $(\exists x : t)\varphi$ in Def. 10 can be stated as $(\exists\, x' : \widehat{\tau_T^{T'}}(t))\big(((\exists x : t)\overline{repr}(x, x')) \wedge Tr_\sigma(\varphi)\big)$.

Another restriction that needs to be relaxed in the definitions above, is the totality of the maps. It is not unusual that different formalisations, responding to different needs, only specify the portion of the language signature, i.e., types,

```
repr(f:fseq,l:list): bool =
  length(l) = f`length AND
    FORALL(i:nat): i < length(l)
      IMPLIES nth(l,i) = f`seq(i)
```

```
empty?_fseq_homomorphic: LEMMA
  FORALL(f:fseq,l:list|repr(f,l)):
    empty?_fseq(f) IFF null?(l)
```

Fig. 2. PVS implementation of the $\overline{repr}$ for the example of Fig. 1 (*left*) and the lemma about its homomorphism w.r.t. the `empty?_fseq` function (*right*).

constants, functions and predicates, required by the context where that particular specification is used. This observation means that the approach presented before should be able to cope with partial language signature maps.

From a practical point of view, a possible way to support this feature with minimum impact in the definitions above is to restrict the sets of syntactic symbols taken into consideration in the representation process. First, note that not every type in the source theory is needed in the context of the target theory. For example, when constructing the representation of `fseqs` using `list` elements, the type `barray` is just an auxiliary concept that needs no counterpart on the `list` side. Then, the domain of the intended type map $\widehat{\mathcal{T}^{\mathtt{list}}_{\mathtt{fseq}}}$ should be $\mathit{Type}(T_{\mathtt{fseqs}}) \setminus \mathtt{barray}$. Secondly, the domain of the total functions $\sigma^{\mathcal{C}}$, $\sigma^{\mathcal{F}}$ and $\sigma^{\mathcal{P}}$ in the language signature map (Def. 8) would be a subset of the whole set of constant (resp. function and predicate) symbols of the source theory presentation. In the example, the domain of $\sigma^{\mathcal{C}}$ should be $\mathcal{C}_{\mathtt{fseqs}} \setminus \{\mathtt{default}\}$. Finally, the term, formula, and theoroidal representation relation is now restricted to accept those terms and formulas that can be constructed using only the symbols of interest. It is important to note that this restriction on the symbols could provoke that some proofs from the source theory can not be preserved in the target theory. This occurs when terms or formulas that can only be constructed using discarded symbols are explicitly provided during the application of a proof step, such as in the introduction of a new hypothesis (cut rule, for example) or in the instantiation of existential-strength quantifiers. While there is no automatic way to solve this kind of problem, it is easily and even mechanically discoverable.

When language signatures are analysed in detail, it is possible to recognize that function and predicate symbols play different roles depending on their logical definition. Some symbols may be axiomatized defining the result of their application to terms and performing observations of their properties that cannot be obtained by other means. Some other symbols may be defined as the composition of other functions (resp. predicates) leading to conservative extensions [15, Sec. 2.3.3] of a theory presentation where such a symbol does not exists. The representation of a symbol in the latter group can be automatically generated whenever the representation of those symbols in the former group have already been provided. In such case, the result of the language signature map applied to the constituent parts of the symbol's definition can be used to extend the language signature map to apply to both sets of symbols. Additionally, the proof that the representation of the symbol does not invalidate the correctness of the term representation relation (second item in Definition 9) can be automatically constructed. It should proceed as a proof by induction on the complexity of the term. This technique is particularly useful when defining representations for algebraic datatypes, since once representations for the constructors, selectors, and recognizers are defined, the rest of the definitions are stated in terms of them or some other symbol whose definition relies on them. Thus, the representation of such symbols can be mostly automated.

Like most modern proof-assistance environments, PVS provides a variety of features intended to help the user in writing complex formalizations. Some of these features include the ability of structuring specifications through the use of specific clauses (`EXPORTING` and `IMPORTING`) and the definition of theory schemas through the use of theory parameters, among others. While specific uses for both characteristics are mentioned above, they can also be used for different purpose. For instance, the `IMPORTING` clause can be used to extend a PVS theory for which a representation is already defined. In such case, the theoretical notion of *theory extension* needs to be further studied in order to establish how the theoroidal representation is affected by this relationship between theories. Theory extensions can be formalized by considering a special kind of (axiom preserving) theoroidal representations relying on language signature maps whose components are injective functions, and term representation relations that are total and one-to-one. These conditions force the formula representation to be analogue to an injective translation, forcing the target theory to extend the source one. *Conservativity* can also be posed in terms of conditions imposed to language signature maps and term representation relations.

The use of theory parameters to define theory schema is one of the most useful PVS features regarding the development of formalizations. When the theory parameters are not part of the representation, i.e., when they are trivially represented, no further consideration is needed. Such is the case of the `fseqs` and `list` example presented in the previous section. However, the theory parameters can also be represented in a non-trivial way in the target theory, as illustrated by the case study presented in the next Section. To formally cope with the impact that both features could impose on the proposed technique, a complete study of the ways theory presentations can be related is necessary, but left as further work.

4 Case Study

Polynomials are widely used to provide smooth approximations of non-linear functions. At the beginning of the last century, Bernstein developed a novel way to represent polynomials in his proof of the StoneWeierstrass approximation theorem [16]. This representation has proved to be specially useful in the field of computerized graphics. Muñoz and Narkawicz [17] developed a formalization based on Bernstein polynomials that can be effectively used to find minimum and maximum values for arbitrary polynomial expressions. Such formalization, provided as a PVS specification available as part of the NASA PVS Library, relied on the fragment of PVS that can be soundly evaluated [8].

```
Polynomial: TYPE = [nat->Coefficient]
Polyproduct: TYPE = [nat->Polynomial]
MultiPolynomial : TYPE = [nat->Polyproduct]

mpoly_eval(bspoly,degmono,cf,m,n)(X) : real =
 sigma(0,n-1,LAMBDA(i:nat):cf(i)*pprod_eval(bspoly(i),degmono,m)(X))
```

Fig. 3. Excerpt from the original multipolynomial formalization part of the Bernstein development.

As part of the upgrade from the version 5 to 6 of PVS, the internal implementation of some PVS data structures was changed. While this change improved the overall performance of the system, it also affected the ground evaluation used in proof strategies developed as part of the Bernstein development. To overcome this problem, it was necessary to change the way in which polynomials were modeled in the formalization. Because of this, the Bernstein development and its strategies were not originally ported from PVS 5 to PVS 6. Recently, the technique proposed in this paper was applied and a new version of the Bernstein algorithms were developed in just a fraction of the time originally estimated. In this section, the case study of the representation of Bernstein polynomials is presented.

The Bernstein development was built around a formalization of multivariate polynomials, or *multipolynomials* for short. Any multipolynomial in m variables (denoted by P below) can be seen as a sum of a finite number, say n, of products between a real coefficient c_i and a so-called *polyproduct*, which is a product of univariate polynomials ($p_{i,j}$).

$$P(x_1, \cdots, x_m) = \sum_{i=1}^{n} c_i \prod_{j=1}^{m} p_{i,j}(x_j) \tag{1}$$

In the first version of the Bernstein development, multipolynomials were modeled using arrays, i.e., functions from natural numbers into a type, as shown in Figure 3. Every index k of a `Polynomial` array p provides access to the coefficient of the k-th power in the univariate polynomial p. Indices i and j in Equation 1 would be used to access `Multipolynomial` and `Polyproduct` arrays respectively. The type `Coefficient` is a renaming for `real`, the PVS type denoting real numbers. Figure 3 also shows the formalization of the evaluation function `mpoly_eval`. Its parameters are respectively: the multipolynomial to be evaluated, the degree of the polynomials in each polyproduct, the coefficients c_i and the values m and n from Equation 1; `X` represents the variables $x_1, \cdots, x_m$. The function `pprod_eval`, omitted for brevity, is the function used to evaluate polyproducts and it is formalized similarly to `mpoly_eval`.

The way in which arrays are used in the algorithms defined as part of the Bernstein development are not amenable for evaluation in PVS. However, only a finite prefix of every array is really used. Therefore, a new formalization is proposed where the array prefixes are represented as finite lists. The additional difficulty of this case with respect to the example of Figure 1 is that `MultiPolynomial` is in fact formalized by a nesting of arrays. Thus, the proposed representation is to be applied at each level of this nesting in order to mimic every type in Figure 3 with the types depicted in Figure 4.

Consequently, this representation of arrays as finite lists is formalized in PVS as shown in Figure 5. The theory `arrays_into_lists`, in fact, uses finite lists of elements of a given unrestricted type (`T2`) to represent arrays containing elements of a possibly different type (`T1`). The relation between the type of contained elements `T2` and `T1` needs to be explicitly stated by a corresponding representing function, `inner_repr` in the figure. Once these three formal elements (`T1`,`T2`, and `inner_repr`) are provided to `arrays_into_lists` as theory parameters, the representation relation between arrays and finite lists is stated in the `repr` relation: a list l represents an array if every element of the list represents the element in the corresponding position of the array.

Note that the proposed representation, in contrast to the example of the previous section, is not injective. Each list l of elements of type `T2` can be used to represent any of an infinite number of arrays starting with `T1` elements in the order induced by l. For instance, the empty list can be used to represent any array. This coarse-grain property of the representation has to be taken care of by the theory using it to establish the representation relation between `MultiPolynomial` and `MultiPolynomialList`.

Figure 6 shows such a theory. The theory parameters are, respectively, two natural numbers representing the number of terms in the sum and the degree of the polynomial (n and m in Equation 1) and the degree of the polynomials in each polyproduct. Note how every importing of the `arrays_into_lists` theory is adequately instantiated using, at each nesting level, the representation of the nested type stated by the previous importing clause. Then, the representation relation states that a `MultipolynomialList` represents a `MultiPolynomial` if for every list in `MultipolynomialList` at every nesting level: (*1*) it represents the corresponding array in `MultiPolynomial` according to the `arrays_into_list` theory and (*2*) its length is correct according to the theory parameters. This latter condition assures that the lists being used to represent the `MultiPolynomial` are exactly those that have to be used. Finally, the equivalence between evaluation functions of both formalizations is stated and proved. The proof proceeds by exploring symmetrically the structure of both functions and leveraging the equivalence lemmas from the auxiliary functions.

```
PolynomialList : TYPE = list[Coefficient]
PolyproductList : TYPE = list[PolynomialList]
MultiPolynomialList : TYPE = list[PolyproductList]

a2l__mpoly_eval(bsplist,degmono,cf,m,n)(X) : real =
 sigma(0,n-1,LAMBDA(i:nat):cf(i)*pprod_eval(nth(bsplist,i),degmono,m)(X))
```

Fig. 4. Excerpt from the new formalization for multipolynomials based on lists.

```
arrays_into_lists[T1,T2: TYPE, inner_repr: [T1,T2->bool]]: THEORY BEGIN
  repr(A:[nat->T1], l:list[T2]): bool =
   FORALL(i:below(length(l))): inner_repr(A(i), nth(l,i))
END arrays_into_lists
```

Fig. 5. PVS theory for the representation of arrays using finite lists.

```
multipoly_into_polylist[n,m:posnat, degmono: [nat->nat]]: THEORY BEGIN
  IMPORTING
    arrays_into_lists[Coefficient,Coefficient,=] AS polynomial,
   arrays_into_lists[Polynomial,PolynomialList,polynomial.repr]
    AS polyproduct,
   arrays_into_lists[Polyproduct,PolyproductList,polyproduct.repr]
    AS multipolynomial

  repr(mp: MultiPolynomial, pl: MultipolynomialList): bool =
    multipolynomial.represents(pl,mp) AND
    n = length(pl) AND
    (FORALL (pp_i: below(n)): m = length(nth(pl,pp_i))) AND
    (FORALL (pp_i: below(n), var_i: below(m)):
     length(nth(nth(pl,pp_i),var_i)) = degmono(var_i) + 1)

  a2l__multibs_eval__equivalence: LEMMA
   FORALL(mp: MultiPolynomial, pl: MultiPolynomialList):
    repr(mp,pl) IMPLIES
     FORALL(X: Vars, cf: [nat->real]):
      multibs_eval(mp,dm,cf,m,n)(X) = a2l__multibs_eval(pl,dm,cf,m,n)(X)

END multipoly_into_polylist
```

Fig. 6. Representation of multipolynomials on arrays using mulitpolynomials as lists.

New versions of the Bernstein algorithms and proof strategies were also defined, changing only the few places where explicit references to the datatype representing multipolynomials were found. As the correctness theorems for such algorithms were expressed in terms of the evaluation of the multipolynomial, the equivalence lemma `a2l__multibs_eval__equivalence` was used to easily show that all the algorithms on `MultipolynomialList` are also correct. The proposed translation of multipolynomials had the direct impact of providing an executable and proven correct version of the Bernstein algorithms in a small fraction of the time it was estimated just for fixing the previous version. Moreover, the new version outperformed the previous one. This is the case because lists are translated to Lisp in a much more efficient way than arrays, and because the original definition also performed some internal conversions to lists that are unnecessary in the new version.

5 Related Work

The idea of establishing formal connections between datatypes has been approached from different flanks and support for related features has been added to a variety of formal systems. Notably, much of the effort has been posed on the connection between isomorphic datatypes. The goal of the technique presented in this paper is more general. Since its main motivation emerged from the practical problem of facilitate the reuse of existent formalizations, limiting the scope of application to isomorphic types would be too restrictive. The downside of such design decision is that automatization is harder to achieve.

For the sake of brevity, the universe of comparison explored in the following is restricted to the context of higher-order logic environments, such as Isabelle [18] and Coq [19]. In such context, the technique presented in this paper is closely related to the formalization of quotients in Isabelle [20]. There, the connection between a so-called *raw* type and a more abstract type is established through the *lifting* of terms from the raw to the abstract type and the *transfer* of theorems between them. Besides the similarities, there are some important differences. First, conceptually the work on quotients appears to be designed with a directionality in mind. While this directionality is not explicit in the transfer-related features, since theorems can be transfered from raw to abstract and vice versa, such notion is still present in the lifting functionalities. On the other hand, in the technique presented in this paper the roles of abstract and raw type are not forced explicitly. Indeed, the representation presented in this paper is not just about abstraction (or concretization) relations, but it allows for more general relationships between types of different nature. More concretely, the lifting of terms described in [20] is based on the existence of two functions, *Abs* and *Rep*, that relate an abstract instance with its raw counterpart. Meanwhile, the corresponding relations in the technique proposed in this paper are not necessarily functional. This feature makes it easy to apply the proposed technique to the example presented in the Section 4. Nevertheless, there is a cost associated: the level of automation described in [20] is not reachable, in general, for the proposed technique. For particular cases, as those identified in the NASA PVS Library, automation seems feasible and it is work in progress.

Also addressing the problem of transfering theorems along isomorphic types, Zimmermann and Herbelin [21] have proposed a technique for Coq that shares much of the spirit of the one presented here. Nevertheless, similarly to [20], their work also relies in a functional notion to relate elements from different datatypes. Again, while automation is easier to achieve, in fact a simple algorithm to translate proofs is explicitely presented in the mentioned paper, the use of a relational representation mechanism makes the technique presented here applicable to cases that can not be addressed in functional representation settings. Additionally, the transfer mechanism in [21] does not support some features already supported in this paper, such as the compositionality of the technique showed in the case study. Other similar approaches have been proposed in the context of the Coq system. For example, the concept of signature for higher-order functions by Sozeau [22] resembles the idea of transfer of theorems proposed in this paper. Furthermore, the work of Magaud [23] on translation of proof terms for Coq also addressed the problem of minimizing the effort in sharing theorems of related datatypes. Both approaches are less general than the work presented in this paper.

Regarding PVS, the system provides native suport for theory interpretations [24]. This feature allows for the instantiation of uninterpreted symbols such as constant, function, and even type symbols. The necessary proof obligations are automatically generated by the type-checker in order to ensure the validity of the interpretation. While this feature greatly improves reusability of theories, the examples described in this paper fall out of its scope of application because interpretations are limited to refinement of uninterpreted symbols.

A typical instantiation of the problem of formal connections is the refinement of abstract into concrete datatypes. In this practical but more restricted case, elaborated features can be developed and a higher level of automation can be achieved. Notable examples of refinement in higher-order logic systems were developed by Lammich [25] for Isabelle/HOL and Cohen et. al. [26] for Coq, among others.

There are also similar techniques that were specially developed with the aim to connect different dependent types. From the observation that when working with dependent types is not unusual to find cases were the only difference between them is given by its *logical* description, while the structural definition is almost or directly the same, McBride have developed the notion of *Ornaments* [27], which allows to handle type declarations as first-order citizens of the formal setting and establish relationships between them. These relationships are particularly aimed at qualifying one of the types as more informative than the other. Ornaments are specially designed to relate inductive structures, for instance, natural numbers (defined inductively using zero and successor) and finite-length lists. On the contrary, the technique presented in this paper does not suffer from such restriction: the (possible dissonant) nature of the datatypes being connected does not limit the applicability of the connection process. Somehow in the same line of Ornaments but motivated by the extraction of code from a formally verified description, Dagand et. al. have presented a technique grounded on an application of the notion of Galois connections that outperforms Ornaments in that it supports the connection between more general datatypes [28]. The constructive setting in which such technique is developed makes it very different from the approach explained in these pages.

6 Conclusion

This paper presents a formal study of the concept of connections between theory presentations in a higher-order logic setting. The concept is approached in a way as general as possible in order to maximize the range of application of the representation technique. In particular, the proposed approach does not only apply to refinement of abstract into concrete types, but also to more general relationships between types. A non-trivial case study is presented to illustrate the usefulness of the proposed technique.

While the technique is not implemented as an automatic procedure yet, the systematic nature of the approach hints that automation can be achieved for a considerable part of the process. For specific cases, such as when dealing with structured and algebraic datatypes, the automation of a significant part of the technique is planned to be undertaken as future work. On the theoretical side, further research is needed to understand how the technique presented in this paper is affected by the various theory extension mechanisms provided by modern proof assistants.

Acknowledgments. Research by the first, second, and forth author was supported by the National Aeronautics and Space Administration under NASA/NIA Cooperative Agreement NNL09AA00A. Research by the second author was also supported by the *Agencia Nacional de Promoción Científica y Tecnológica* (ANPCyT) under grant PICT 2013-2129 and by the *Concejo Nacional de Investigaciones Científicas y Técnicas* (CONICET) under grant PIP 11220130100148CO.

References

1. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In Kapur, D., ed.: Proceedings of the 11th. International Conference on Automated Deduction (CADE). Volume 607 of Lecture Notes in Artificial Intelligence., Saratoga, NY, Springer-Verlag (June 1992) 148–752
2. Burris, S., Sankappanavar, H.P.: A course in universal algebra. Graduate Texts in Mathematics. Springer-Verlag, Berlin, Germany (1981)
3. Enderton, H.B.: A mathematical introduction to logic. Academic Press (1972)
4. van Benthem, J., Doets, K.: Higher-order logic. In Gabbay, D., Guenthner, F., eds.: Handbook of Philosophical Logic. Volume 1. second edn. Kluwer Academic Publishers (2001) 275–329
5. Troelstra, A.S., Schwichtenberg, H.: Basic Proof Theory. Number 43 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (1996)
6. Girard, J.Y., Lafont, Y., Taylor, P.: Proofs and types. Number 7 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (1989)
7. Barendregt, H.P.: Lambda calculi with types. In Abramsky, S., Gabbay, D., Maibaum, T.S.E., eds.: Handbook of Logic in Computer Science. Volume II. Oxford University Press (1999)
8. Muñoz, C.: Rapid prototyping in PVS. Contractor Report NASA/CR-2003-212418, NASA, Langley Research Center, Hampton VA 23681-2199, USA (May 2003)
9. Ehrig, H., Mahr, B., Orejas, F.: Introduction to algebraic specification. Part 1: Formal methods for software development. Computer Journal **35**(5) (October 1992) 468–477
10. Ehrig, H., Mahr, B., Orejas, F.: Introduction to algebraic specification. Part 2: From classical view to foundations of system specifications. Computer Journal **35**(5) (October 1992) 468–477
11. McLane, S.: Categories for working mathematician. Graduate Texts in Mathematics. Springer-Verlag, Berlin, Germany (1971)
12. Pierce, B.C.: Basic category theory for computer scientists. MIT Press (1991)

13. Meseguer, J.: General logics. In Ebbinghaus, H.D., Fernandez-Prida, J., Garrido, M., Lascar, D., Artalejo, M.R., eds.: Proceedings of the Logic Colloquium '87. Volume 129., Granada, Spain, North Holland (1989) 275–329
14. Goguen, J.A., Roşu, G.: Institution morphisms. Formal Aspects of Computing **13**(3-5) (2002) 274–307
15. Turski, W.M., Maibaum, T.S.E.: The specification of computer programs. International Computer Science Series. Addison–Wesley Publishing Co., Inc. (1987)
16. Bernstein, S.: Démonstration du théorème de weierstrass fondée sur le calcul des probabilités. Communications of the Kharkov Mathematical Society **13**(1) (1912) 1–2
17. Muñoz, C., Narkawicz, A.: Formalization of a representation of Bernstein polynomials and applications to global optimization. Journal of Automated Reasoning **51**(2) (August 2013) 151–196
18. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
19. Bertot, Y., Castéran, P.: Interactive theorem proving and program development: CoqArt: the calculus of inductive constructions. Springer Science & Business Media (2013)
20. Huffman, B., Kunčar, O.: Lifting and transfer: A modular design for quotients in isabelle/hol. In Gonthier, G., Norrish, M., eds.: Certified Programs and Proofs, Cham, Springer International Publishing (2013) 131–146
21. Zimmermann, T., Herbelin, H.: Automatic and transparent transfer of theorems along isomorphisms in the coq proof assistant. arXiv preprint arXiv:1505.05028 (2015)
22. Sozeau, M.: A new look at generalized rewriting in type theory. Journal of Formalized Reasoning **2**(1) (2010) 41–62
23. Magaud, N.: Changing data representation within the Coq system. In: International Conference on Theorem Proving in Higher Order Logics, Springer (2003) 87–102
24. Owre, S., Shankar, N.: Theory interpretations in PVS. Technical Report SRI-CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park, CA (2001)
25. Lammich, P.: Refinement based verification of imperative data structures. In: Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs. CPP 2016, New York, NY, USA, ACM (2016) 27–36
26. Cohen, C., Dénès, M., Mörtberg, A.: Refinements for free! In Gonthier, G., Norrish, M., eds.: Certified Programs and Proofs, Cham, Springer International Publishing (2013) 147–162
27. McBride, C.: Ornamental algebras, algebraic ornaments. Unpublished (2010)
28. Dagand, P.É., Tabareau, N., Tanter, É.: Foundations of dependent interoperability. Journal of Functional Programming **28** (2018)