Encoding of Predicate Subtyping and Proof Irrelevance in the $\lambda\Pi$-calculus Modulo Theory

Gabriel Hondet
Frédéric Blanqui
Université Paris-Saclay, ENS Paris-Saclay, CNRS, INRIA
Laboratoire Spécification & Vérification, 91190, Gif-sur-Yvette, France

--- Abstract ---

The $\lambda\Pi$-calculus modulo theory is a logical framework in which various logics and type systems can be encoded, thus helping the cross-verification and interoperability of proof systems based on those logics and type systems. In this paper, we show how to encode predicate subtyping and proof irrelevance, two important features of the PVS proof assistant. We prove that this encoding is correct and that encoded proofs can be mechanically checked by Dedukti, a type checker for the $\lambda\Pi$-calculus modulo theory using rewriting.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Higher order logic; Theory of computation → Equational logic and rewriting

Keywords and phrases Predicate Subtyping, Logical Framework, PVS, Dedukti, Proof Irrelevance

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

A substantial number of proof assistants can be used to develop formal proofs, but a proof developed in an assistant cannot, in general, be used in another one. This impermeability generates redundancy since theorems are likely to have one proof per proof assistant. It also prevents adoption of formal methods by industry because of the lack of standards and the difficulty to use adequately formal methods.

Logical frameworks are a part of the answer. Because of their expressiveness, different logics and proof systems can be stated in a common language. The $\lambda\Pi$-calculus modulo theory, or $\lambda\Pi/\equiv$, is such a logical framework. It is the simplest extension of simply typed $\lambda$-calculus with dependent types and arbitrary computation rules. Fixed-length vectors are a common example of dependent type, that can be represented in the $\lambda\Pi$-calculus as $\forall n : \mathbb{N}, \text{Vec}(n)$. The $\lambda\Pi$-calculus modulo theory already allows to formulate first order logic, higher order logic [4] or proof systems based on Pure Type Systems [11] such as Matita [2], Coq [9] or Agda [15].

PVS [27] is a proof assistant that has successfully been used in collaboration by academics and industrials to formalise and specify real world systems [26]. More precisely, PVS is an environment comprising a specification language, a type checker and a theorem prover. One of the specificities of PVS is its ability to blend type checking with theorem proving by requiring terms to validate arbitrary predicates in order to be attributed a certain type. This ability is a consequence of predicate subtyping [29]. It facilitates the development of specifications and provides a more expressive type system which allows to encode more constraints. For instance, one can define the inverse function $\text{inv} : \mathbb{R}^* \rightarrow \mathbb{R}$, where $\mathbb{R}^*$ is a predicate subtype defined as reals which are not zero.

If predicate subtyping provides a richer type system, it also makes type checking of specifications undecidable. In [16], F. Gilbert paved the way of the expression of PVS into $\lambda\Pi/\equiv$: he formalised the core of PVS and provided a language of certificates for PVS whose type checking is decidable. However, the encoding in $\lambda\Pi/\equiv$ of this language of certificates...
The following work proposes an encoding of proof irrelevant equivalences into the \(\lambda \Pi\)-calculus modulo theory. It also inspects the completion of such equations into a confluent rewrite system. The resulting rewrite system can be used to provide an encoding of PVS into DEDUKTI, a type-checker for the \(\lambda \Pi\)-calculus modulo theory based on rewriting [3].

Related work

An encoding, or “simulation” of predicate subtyping à la PVS into HOL can be found in [19]. The objective of that work was to get some facilities provided by predicate subtyping into HOL rather than providing a language of certificates, and proof checking hence remains undecidable. Moreover, predicate subtypes are not represented by types but by theorems.

In [31], predicate subtyping is weakened into a language named RUSSELL to be then converted into CIC. This conversion amounts to the insertion of coercions and unsolved meta-variables, the latter embody PVS type correctness conditions (TCC). The equational theory used in the CIC encoding is richer than ours since it includes surjective pairing \(e = \text{pair} TU (\text{fst} TU e) (\text{snd} TU e)\) and \(\eta\)-equivalence \(f = \lambda x . f x\) in addition to proof irrelevance.

In [33], proof irrelevance is embedded into Luo’s ECC [24] and its dependent pairs. Pairs and dependent pair types come in two flavours, the proof irrelevant one and the normal one. The flavour is noted by an annotation, and proof irrelevance is implemented by a reduction which applies only on annotated pairs. The article presents as well an application to PVS.

On a slightly more practical side, the automated first-order prover ACL2 [20] reproduces the system of “guards” provided by predicate subtyping into its logic based on COMMON LISP with the concept of gold symbols. Approximately, a symbol is gold if all its TCC have been solved.

Some theories—often based on Martin-Löf’s Type Theory—blend together a decidable (called definitional or intensional) equality with an undecidable (said extensional) equality. In [28], a judgement “\(A\) is provable” is introduced, to say that a proof of \(A\) exists, but no attention is paid to what it is. In the same context, article [30] provides proof irrelevance for predicate subtyping (here called subset types) for two different presentations, one is intensional, and the other extensional. The interested reader may have a look at NUPRL [10], an implementation of Martin-Löf’s Type Theory with extensional equality and subset types.

Proof irrelevance has also been added to LF to provide a new system LFI in [23], where proof irrelevance is used in the context of refinement types. In LFI, proof irrelevance is not limited to propositions, nor it is attached to a certain type: terms are irrelevant based on the function they are applied to. A similar system is implemented in AGDA [32].

More generally, concerning proof irrelevance in proof assistants, COQ and AGDA [17] each have a sort for proof irrelevant propositions (SProp for COQ and Prop for AGDA [32]). LEAN [13] is by design proof irrelevant, and MATITA supports proof irrelevance as well [1, section 9.3].

Outline

Encoding predicate subtyping requires a clear definition of it, which is done in Section 2. Predicate subtyping is encoded into \(\lambda \Pi/\equiv\) using the signatures provided in Section 3. This encoding is put in use into some examples as well. The encoding is proved correct in Section 4: any well typed term of the source language can be encoded into \(\lambda \Pi/\equiv\), and its type in \(\lambda \Pi/\equiv\) is the encoding of its type in the source language. Finally, we show that a type checker for
the λΠ-calculus modulo rewriting can be used to type check terms that have been encoded as described in Section 3.

2 PVS-Cert: A Minimal System With Predicate Subtyping

Because of its size, encoding the whole of PVS cannot be achieved in one step. Consequently, F. Gilbert in his PhD [16] extracted, formalised and studied a subsystem of PVS which captures the essence of predicate subtyping named PVS-Cert. Unlike PVS, PVS-Cert contains proof terms, which has for consequence that type checking is decidable in PVS-Cert while it is not in PVS. Hence PVS-Cert is a good candidate to be a logical system in which PVS proofs and specifications can be encoded to be rechecked by external tools.

In this paper, we will use an equational presentation of PVS-Cert. We will hence use equations rather than reduction rules and slightly change the syntax of terms. We describe PVS-Cert, as done in [16], namely the addition of predicate subtyping over simple type theory.

2.1 Type Systems Modulo Theory

To describe PVS-Cert and λΠ/≡ in a uniform way, we will use the notion of Type Systems Modulo described in [7]. Type Systems Modulo are an extension of Pure Type Systems [6] with symbols of fixed arity whose types are given by a typing signature Σ, and an arbitrary conversion relation ≡ instead of just β-conversion ≡β.

The terms of such a system are characterised by a finite set of sorts S, a countably infinite set of variables V and a signature Σ. The set of terms ℱ(Σ, S, V) is inductively defined in Figure 1.

\[
M, N, T, U ::= s \in S \mid x \in V \mid M \downarrow N \mid \lambda x : T, M \mid (x : T) \rightarrow U \mid f \overrightarrow{M}
\]

\[
\text{with } \Sigma(f) = (\overrightarrow{x}, T, U, s)
\]

Figure 1 Terms of the type system characterised by S, V and Σ.

The contexts are noted \(\Gamma ::= \emptyset \mid \Gamma, v : T\) and the judgements \(\Gamma \vdash WF\) or \(\Gamma \vdash M : T\). The typing rules are defined in Figure 2 and depend on:

- axioms \(A \subseteq S \times S\) to type sorts;
- product rules \(P \subseteq S \times S \times S\) to type dependent products;
- a typing signature Σ which defines the function symbols and how to type their applications;
- a convertibility relation ≡.

Remark 1 (Notations). Rewriting relations are noted \(\leftrightarrow_R\), where \(R\) is a set of rewriting rules. \(\leftrightarrow_R\) is the closure of \(R\) by substitution and context. \(\equiv_R\) is the symmetric, reflexive and transitive closure of \(\leftrightarrow_R\). The substitution of \(x\) by \(N\) in \(M\) is noted \(\{x \mapsto N\} M\). We use a vectorised notation for products \((\overrightarrow{x} : T) \rightarrow U\) to represent the dependent product \((x_1 : T_1) \rightarrow (x_2 : T_2) \rightarrow \cdots (x_n : T_n) \rightarrow U\); and more generally for any construction that can be extended to a finite sequence, such as a parallel substitution \(\{x \mapsto \overrightarrow{N}\} M\). A mapping \(\Sigma(f) = (\overrightarrow{x} : T, U, s)\) can also be written \(x : T \overrightarrow{\Sigma} f(\overrightarrow{x}) : U : s\).
Encoding of Predicate Subtyping and Proof Irrelevance in the \( \lambda \Pi \)-calculus Modulo Theory

\[
\begin{align*}
\text{empty} & \quad \emptyset \vdash WF \\
\text{decl} & \quad \Gamma \vdash T : s \quad \forall v \notin \Gamma \\
\text{var} & \quad \Gamma, v : T \vdash WF \\
\text{sort} & \quad \Gamma \vdash WF \\
\text{prod} & \quad \Gamma \vdash T : s_1, x : T \vdash U : s_2 \quad (s_1, s_2, s_3) \in \mathcal{P} \\
\text{abst} & \quad \Gamma \vdash (x : T) \to U : s_3 \\
\text{app} & \quad \Gamma \vdash M : (x : T) \to U \quad \Gamma \vdash N : T \Rightarrow \Gamma \vdash M \cdot (x \mapsto N) : U \\
\text{sig} & \quad \Gamma \vdash f(\tilde{t}) : \{x \mapsto \tilde{t}\} \cdot U
\end{align*}
\]

\[\Sigma(\hat{f}) = (\tilde{x}, T, U, s)\]

\[\Gamma \vdash f(\tilde{t}) : \{x \mapsto \tilde{t}\} \cdot U\]

Figure 2 Typing rules of a Type System Modulo.

2.2 Simple Type Theory

PVS and PVS-Cert are both based on simple type theory, which can be represented by the PTS \( \lambda \text{HOL} \) [6]:

\[
\begin{align*}
\mathcal{S}_{\lambda \text{HOL}} & = \{\text{Prop}, \text{Type}, \text{Kind}\}, \\
\mathcal{A}_{\lambda \text{HOL}} & = \{(\text{Prop}, \text{Type}), (\text{Type}, \text{Kind})\}, \\
\mathcal{P}_{\lambda \text{HOL}} & = \{(\text{Prop}, \text{Prop}, \text{Prop}), (\text{Type}, \text{Type}, \text{Type}), (\text{Type}, \text{Prop}, \text{Prop})\}, \\
\Sigma_{\lambda \text{HOL}} & = \emptyset, \\
\equiv_{\lambda \text{HOL}} & \text{ is the reflexive, transitive and symmetric closure of the } \beta\text{-equation}
\end{align*}
\]

\[\equiv_{\lambda \text{HOL}} \quad ((\lambda x, M) \cdot N) = \{x \mapsto N\} \cdot M\]  \((\beta)\)

2.3 Predicate Subtyping

Predicate subtyping has two main benefits for a specification language. The first is to provide a richer type system thanks to the entanglement of type-checking and proof-checking. In consequence, any property can be encoded in the type system, which allows to easily create "guards" such as \texttt{tail : nonempty_stack → stack} where \texttt{nonempty_stack} is a predicate subtype defined from a predicate \texttt{empty}?. It is also essential in the expression of mathematics: the judgement \( M : T \) is akin to the statement \( M \in T \) in the usual language of mathematics when \( T \) is a set defined by comprehension such as \( E = \{n : \mathbb{N} \mid P(n)\} \) where \( P(n) = (n^2 = 1 \mod 2) \). With predicate subtyping, we can represent the set \( E \) by the type \texttt{psub N P}, and the judgement \( \Gamma \vdash M : \text{psub N P} \) is derivable if term \( M \) contains a proof of \( P(n) \) for some \( n \).

The other benefit of predicate subtyping, which is essential in PVS developments, is that it allows to separate the process of writing a specification from that of proving a theorem. In PVS, this separation appears through \textit{type correctness conditions} (TCC): the development of a specification creates proof obligations that may be solved at any time. This separation is also visible in usual mathematical developments, where if we want to prove that \( t \in E \), we prove once that \( t = 1 \mod 2 \) to then forget the proof and simply use \( t \).

The type system of PVS-Cert can be seen as \( \lambda \text{HOL} \) with a non empty signature \( \Sigma_{\text{PVS}} \) defined in Figure 3 and a richer equivalence \( \equiv \) that will be discussed in the next paragraph.
A predicate subtype \( \text{psub}TU \) is defined from a supertype \( T \) and predicate \( U \) which binds a variable of type \( T \) to a proposition. Terms inhabiting a predicate subtype \( \text{psub}TU \) are built with the pair construction \( \text{pair}TUMN \) where \( M \) is a term of the supertype \( T \) and \( N \) is a proof of \( (UM) \). While the pair construction allows to coerce a term from any type to a predicate subtype, the converse, that is the coercion from a type to its supertype is done with \( \text{fst} \), the left projection of the pair. The right projection, \( \text{snd} \), provides a witness that the left projection of the pair validates the predicate defining the subtype.

Equations and Proof Irrelevant Pairs

The equivalence of PVS-Cert is noted \( \equiv_pvs \) and contains Equations (5), (6) and (\( \beta \)) which provide proof irrelevance in pairs.

\[
\begin{align*}
\text{pair } t \ u \ m \ h_0 & = \text{pair } t \ u \ m \ h_1 \\
\text{fst } t_0 \ u_0 \ (\text{pair } t_1 \ u_1 \ m \ h) & = m
\end{align*}
\]

We will now motivate the use of these equations in PVS-Cert. Proofs contained in terms are essential for typing purposes. On the other hand, these proofs are a burden regarding equivalence of terms. Were these proofs taken into account (as \( \equiv_\beta \) does), too many terms would be distinguished. For example, consider two terms \( t = \text{pair} \mathbb{N} \text{Even}2h \) and \( u = \text{pair} \mathbb{N} \text{Even}2h' \) typed as even numbers. Then \( t \) and \( u \) are not considered equal because they don’t have the same proof \( (h \text{ and } h') \) that 2 is even. We end up with as many even numbers 2 as there are proofs that 2 is even.

As stated in [12], most mathematicians seek convertibility of \( t \) and \( u \) and care more about what \( h \) and \( h' \) prove than the proofs themselves. To this end, PVS-Cert has proof irrelevant pairs: proofs attached to terms are not taken into account when checking the equivalence of two pairs. This property is embedded in the equivalence relation \( \equiv_pvs \) used in the conversion rule of PVS-Cert which must verify Equation (5).

Equation (6) allows the projection to compute, but because of proof irrelevance, we cannot allow the right projection to compute, otherwise, all terms of type \( \text{Prop} \) would be considered equivalent.

We motivate further the need for proof irrelevance with an example which evinces a term that requires proof irrelevance to be type checked.

Example 2 (Proof irrelevance in type checking). Let \( \iota : \text{Type} \), \( \text{Even} : \iota \to \text{Prop} \), \( \text{even} := \text{psub}\iota \text{Even} \) and \( \text{pg}(h) := \text{psub}\iota (g(\text{pair} \iota \text{Even}n \ h)) \). The term \( f u \) in context \( \Gamma = n : \iota; h : \text{Even}n; h' : \text{Even}n; g : \text{even} \to \text{Prop}; f : \text{psub}\iota (\text{pg}(h)); u : \text{psub}\iota (\text{pg}(h')) \) requires proof irrelevance to be proof checked, as can be seen in the following derivation tree (extracted.
from the typing derivation),

\[
\frac{\Gamma \vdash f : \text{pg}(h) \rightarrow t}{\Gamma \vdash fu : t}
\]

The upper right premise requires \( g \) (pair \( \iota \) Even \( n \) \( h \)) to be convertible with \( g \) (pair \( \iota \) Even \( n \) \( h' \)) and hence pair \( \iota \) Even \( n \) \( h \) to be convertible with pair \( \iota \) Even \( n \) \( h' \).

A proof that \( T \equiv \beta U \) or \( T \equiv \rho \) can use untyped intermediate terms, which can be problematic when one wants to prove some property on typed terms only. In the case of \( \equiv \beta \), the problem is solved by using the fact that \( \leftrightarrow \beta \) is confluent, that is \( \equiv \beta = \leftrightarrow \beta \). We now prove a similar property for \( \equiv \rho \):

\[\text{Lemma 3 (Properties of the PVS-CERT conversion). Let } \leftrightarrow \beta_{\text{fst}} = \leftrightarrow \beta \cup \leftrightarrow \beta_{\text{fst}} \text{ where } \leftrightarrow \beta_{\text{fst}} \text{ is the closure by substitution and context of Equation (6) oriented from left to right, and let } \leftrightarrow \rho_{\text{pi}} \text{ be the closure by substitution and context of Equation (5).}
\]

For all relations on terms \( R \) and \( S \), let \( R^S \) be the restriction of \( R \) to typable terms, \( R^* \) be the reflexive and transitive closure of \( R \), and \( RS \) is the composition of \( R \) and \( S \). Then:

\[\equiv_{\rho_{\text{pi}}} \subseteq \equiv_{\beta_{\text{fst}}} \cup \equiv_{\beta_{\text{pi}}} \quad \text{where } \equiv_{\rho_{\text{pi}}} \text{ is the reflexive closure of } \leftrightarrow \rho_{\text{pi}}. \]

Assume that the \( \leftrightarrow \rho_{\text{pi}} \) step is at position \( p \) and the \( \leftrightarrow \beta_{\text{fst}} \) step is at position \( q \). If \( p \) and \( q \) are disjoint, this is immediate. If \( p \) is above \( q \), we have pair \( T \cup M N \) and either pair \( T \cup M N \) or pair \( T \cup M N \) pair \( T' \cup M' \). In the first case, pair \( T \cup M N \) \( \rightarrow \rho_{\text{pi}} \) \( M \). In the second case, pair \( T \cup M N \) \( \rightarrow \beta_{\text{fst}} \) pair \( T' \cup M' \). Finally, if \( q \) is above \( p \), we have \( (\lambda x : T, M) \) \( N \) \( \leftrightarrow \rho_{\text{pi}} \) \( (\lambda x : T', M') \) \( N' \) \( \leftrightarrow \beta_{\text{fst}} \) \( \{ x \mapsto N' \} \). We now prove that \( \equiv_{\beta_{\text{fst}}} \) is confluent modulo \( \rho_{\text{pi}} \), that is, \( \equiv_{\rho_{\text{pi}}} \subseteq \equiv_{\beta_{\text{fst}}} \cup \equiv_{\beta_{\text{pi}}} \).

We now prove that \( \equiv_{\rho_{\text{pi}}} \) steps can be postponed: \( \equiv_{\rho_{\text{pi}}} \subseteq \equiv_{\beta_{\text{fst}}} \cup \equiv_{\beta_{\text{pi}}} \), where \( \equiv_{\beta_{\text{fst}}} \) is the reflexive closure of \( \leftrightarrow \beta_{\text{fst}} \). Assume that the \( \leftrightarrow \rho_{\text{pi}} \) step is at position \( p \) and the \( \leftrightarrow \beta_{\text{fst}} \) step is at position \( q \). If \( p \) and \( q \) are disjoint, this is immediate. If \( p \) is above \( q \), we have pair \( T \cup M N \) and either pair \( T \cup M N \) or pair \( T \cup M N \) pair \( T' \cup M' \). In the first case, pair \( T \cup M N \) \( \rightarrow \rho_{\text{pi}} \) \( M \). In the second case, pair \( T \cup M N \) \( \rightarrow \beta_{\text{fst}} \) pair \( T' \cup M' \). Finally, if \( q \) is above \( p \), we have \( (\lambda x : T, M) \) \( N \) \( \leftrightarrow \rho_{\text{pi}} \) \( (\lambda x : T', M') \) \( N' \) \( \leftrightarrow \beta_{\text{fst}} \) \( \{ x \mapsto N' \} \). We now prove that \( \equiv_{\beta_{\text{fst}}} \) is confluent modulo \( \rho_{\text{pi}} \), that is, \( \equiv_{\rho_{\text{pi}}} \subseteq \equiv_{\beta_{\text{fst}}} \cup \equiv_{\beta_{\text{pi}}} \).

We now prove that \( \equiv_{\rho_{\text{pi}}} \) steps can be postponed: \( \equiv_{\rho_{\text{pi}}} \subseteq \equiv_{\beta_{\text{fst}}} \cup \equiv_{\beta_{\text{pi}}} \), where \( \equiv_{\beta_{\text{fst}}} \) is the reflexive closure of \( \leftrightarrow \beta_{\text{fst}} \). Assume that the \( \leftrightarrow \rho_{\text{pi}} \) step is at position \( p \) and the \( \leftrightarrow \beta_{\text{fst}} \) step is at position \( q \). If \( p \) and \( q \) are disjoint, this is immediate. If \( p \) is above \( q \), we have pair \( T \cup M N \) and either pair \( T \cup M N \) or pair \( T \cup M N \) pair \( T' \cup M' \). In the first case, pair \( T \cup M N \) \( \rightarrow \rho_{\text{pi}} \) \( M \). In the second case, pair \( T \cup M N \) \( \rightarrow \beta_{\text{fst}} \) pair \( T' \cup M' \). Finally, if \( q \) is above \( p \), we have \( (\lambda x : T, M) \) \( N \) \( \leftrightarrow \rho_{\text{pi}} \) \( (\lambda x : T', M') \) \( N' \) \( \leftrightarrow \beta_{\text{fst}} \) \( \{ x \mapsto N' \} \). We now prove that \( \equiv_{\beta_{\text{fst}}} \) is confluent modulo \( \rho_{\text{pi}} \), that is, \( \equiv_{\rho_{\text{pi}}} \subseteq \equiv_{\beta_{\text{fst}}} \cup \equiv_{\beta_{\text{pi}}} \).
We can now conclude as follows. Assume that $A \equiv \text{pvs} \text{ty} B$. By (1), there are $A'$ and $B'$ such that $A \rightarrow_{\beta_{\text{fst}}} A' \equiv_{\text{p}i} B' \rightarrow_{\beta_{\text{fst}}} B$. By (2), (3), (4) and (5), $A(\rightarrow_{\beta_{\text{fst}}}^f) A'(\rightarrow_{\text{p}i}^f) B'(\rightarrow_{\beta_{\text{fst}}}^f) B$. ▶

3 Encoding PVS-Cert in $\lambda\Pi/\equiv$

We provide an encoding of PVS-Cert into the logical framework $\lambda\Pi/\equiv$. This encoding allows to express terms of PVS-Cert into $\lambda\Pi/\equiv$. Because logical frameworks strive to remain minimal, constructions such as pair or psub are not built-in: they must be expressed into the language of the logical framework through an encoding. We hence define the symbols allowing to emulate predicate subtyping using the terms of $\lambda\Pi/\equiv$.

Definition of $\lambda\Pi/\equiv$

$\lambda\Pi/\equiv$ is the family of Type Systems Modulo whose sorts, axioms and product rules are:

- sorts $\mathcal{S}_{\lambda\Pi} = \{ \text{TYPE}, \text{KIND} \}$,
- axiom $\mathcal{A}_{\lambda\Pi} = \{ (\text{TYPE}, \text{KIND}) \}$,
- product rules $\mathcal{P}_{\lambda\Pi} = \{ (\text{TYPE}, \text{TYPE}, \text{TYPE}), (\text{TYPE}, \text{KIND}, \text{KIND}) \}$.

3.1 Encoding Simple Type Theory

The encoding of $\lambda\text{HOL}$ given in Figures 4 and 5 follows the method settled in [11] for pure type systems.

In the following, we write the function symbols of a signature in blue and the other constructions of $\lambda\Pi/\equiv$ in black, to better distinguish them.

The general idea is to manipulate types and terms of $\lambda\text{HOL}$ as terms of $\lambda\Pi/\equiv$. Sorts are both objectified as type and prop and encoded as types by Kind, Type and Prop in Equations (7) to (11). Sorts as types are used to type sorts as objects to encode the axioms in $\mathcal{A}$. Terms of type Type are encoded as terms of type Type. These encoded types can be interpreted as $\lambda\Pi/\equiv$ types with function El (12). Similarly, propositions are reified as terms of type prop and interpreted by function Prf. For instance, given a $\lambda\text{HOL}$ type $T$ and a $\lambda\text{HOL}$ proposition $P$ both encoded as $\lambda\Pi/\equiv$ terms, the abstractions $\lambda x : \text{El} T, x$ and $\lambda h : \text{Prf} P, h$ are valid $\lambda\Pi/\equiv$ terms. The signature exposed in Figure 4 is noted $\Sigma_{\lambda\text{HOL}}$.

Equations (18) to (20) are used to map encoded products to $\lambda\Pi/\equiv$ products. Equation (17) makes sure that the objectified sort prop is the same as the sort Prop when interpreted as a type.

3.2 Encoding Predicate Subtyping

Predicate subtypes are defined in Equation (21) as encoded types (i.e. terms of type Type) built from encoded type t and predicate defined on t. Pairs are encoded in Equation (22), where the second argument is the predicate that defines the type of the pair. The two projections are encoded in Equations (23) and (24), and we note the signature of Figure 6 $\Sigma_{\text{psub}}$.

The signature used to encode PVS-Cert into $\lambda\Pi/\equiv$ is $\Sigma_{\text{PC}} = \Sigma_{\lambda\text{HOL}} \cup \Sigma_{\text{psub}}$. The terms of the encoding are thus the terms of $\mathcal{F}(\Sigma_{\text{PC}}, \mathcal{S}_{\lambda\Pi}, \mathcal{V})$. The typing rules are those of $\lambda\Pi/\equiv$ with the signature $\Sigma_{\text{PC}}$ and the congruence $\equiv$ generated by Equations (5), (6), (17) to (20) and $(\beta)$ where, in Equations (5) and (6), psub, pair and fst (PVS-Cert symbols in black) are replaced by psub, pair and fst ($\lambda\Pi/\equiv$ symbols in blue).
Encoding of Predicate Subtyping and Proof Irrelevance in the $\lambda\Pi$-calculus Modulo Theory

\begin{align*}
\vdash \text{Kind} & : \text{TYPE} : \text{KIND} \quad (7) \\
\vdash \text{Type} & : \text{TYPE} : \text{KIND} \quad (8) \\
\vdash \text{Prop} & : \text{TYPE} : \text{KIND} \quad (9) \\
\vdash \text{type} & : \text{Kind} : \text{TYPE} \quad (10) \\
\vdash \text{prop} & : \text{Type} : \text{TYPE} \quad (11) \\
t & : \text{Type} \vdash \text{El} t & : \text{TYPE} : \text{KIND} \quad (12) \\
p & : \text{Prop} \vdash \text{Prf} p & : \text{TYPE} : \text{KIND} \quad (13) \\
t , p & : \text{Type} , \text{El} t \rightarrow \text{Prop} \vdash \forall t p & : \text{Prop} : \text{KIND} \quad (14) \\
p , q & : \text{Prop} , \text{Prf} p \rightarrow \text{Prop} \vdash p \Rightarrow q & : \text{Prop} : \text{KIND} \quad (15) \\
t , u & : \text{Type} , \text{El} t \rightarrow \text{El} u \vdash t \Rightarrow u & : \text{Type} : \text{KIND} \quad (16)
\end{align*}

\textbf{Figure 4} Signature $\Sigma^{\lambda\text{HOL}}$ of the encoding of $\lambda\text{HOL}$ into $\lambda\Pi/\equiv$.

\begin{align*}
\text{El} \text{prop} & = \text{Prop} \quad (17) \\
\text{Prf}(\forall t p) & = (x : \text{El} t) \rightarrow \text{Prf}(px) \quad (18) \\
\text{Prf}(p \Rightarrow q) & = (h : \text{Prf} p) \rightarrow \text{Prf}(qh) \quad (19) \\
\text{El}(t \Rightarrow u) & = (x : \text{El} t) \rightarrow \text{El}(ux) \quad (20)
\end{align*}

\textbf{Figure 5} Equations of the encoding of $\lambda\text{HOL}$ into $\lambda\Pi/\equiv$.

\section*{3.3 Translation of PVS-Cert Terms Into $\lambda\Pi/\equiv$ Terms}

\begin{definition}[Translation] Let $\Gamma$ be a well formed context. The term translation of the terms $M$ typable in $\Gamma$, noted $[M]_\Gamma$, is defined in Figures 7 and 8.
\end{definition}

\begin{definition}[Type Translation] The type translation of $\text{Kind}$ and the terms $M$ typable by a sort in $\Gamma$, noted $\Gamma[M]_\Gamma$, is defined in Figure 9.
\end{definition}

\begin{definition}[Context Translation] The context translation $\Gamma$ is defined by induction on $\Gamma$ as
\begin{align*}
[\emptyset] & = \emptyset; \\
[\Gamma, x : T] & = [\Gamma], x : [T]_\Gamma.
\end{align*}
\end{definition}

\begin{proposition}[Translation Function] The translation function $[\cdot]$ that maps a context and a PVS-Cert term typable in this context to a $\lambda\Pi/\equiv$ term is well-defined.
\end{proposition}

\begin{proof}
After Lemma 3 and [7, Lemma 41], the types of a term are unique up to equivalence. Moreover, the arguments of the translation function are decreasing with respect to the (strict) subterm relation.
\end{proof}

\section*{3.4 Examples of Encoded Theories}

We provide here some examples that take advantage of proof irrelevance or predicate subtyping. While these examples could have been presented in PVS-Cert, we unfold them into the
encoding of PVS-Cert into λΠ/≡ to show how it can be used in practice. All examples are available as Dedukti files.

**Example 6 (Stacks with predicate subtypes).** This example comes from the language reference manual of PVS [25] and illustrates the use of predicate subtyping and the generation of TCC through a specification of stacks in Figure 10.

Predicate subtyping is used to define the type of nonempty stacks, which allows the function pop to be total. Symbol pop\_push is an axiom that uses Leibniz equality = on stacks. In the definition of the theorem pop2push2, term ?_0 is a meta-variable that must be instantiated with a proof that the first argument of the pair is not empty, and represents, in the encoding, the TCC generated by PVS. We can thus see that the concept of TCC of PVS has a clear and explicit representation in the encoding, allowing its benefits to be transported to λΠ/≡.

**Example 7 (Bounded lists and proof irrelevance).** This example is inspired by sorted lists in the Agda manual [32]². Because we have not encoded dependent types, we cannot encode the type of lists bounded by a variable. We thus declare the bound in the signature. The specification is given in Figure 11.

We first notice that the predicate subtype allows to encode the proof head ≤ bound passed as a standalone argument in Agda in the type of an argument in our encoding, providing a shorter type for bcons. In Figure 12, we define two (non-convertible) axioms p₁ and p₂ as proofs of zero ≤ suc bound, and two lists containing zero but proved to be bounded by suc bound using p₁ for ℓ₁ and p₂ for ℓ₂. Type checking ℓ₁ requires axioms p₁. These axioms are like TCC’s in PVS. Assuming that one wants to prove ℓ₁ ≡ ℓ₂, had we lacked proof irrelevance, we would have had to prove that p₁ ≡ p₂, which is not possible. In our case, the equality is simply the result of refl ℓ₁.

4 Correctness of the Encoding

In this section, we prove that the encoding is correct: if a PVS-Cert type is inhabited then its translation is too. In the following,

s stands for Type, Prop or Kind;

---

1 https://github.com/Deducteam/personojo/paper/
2 https://agda.readthedocs.io/en/v2.5.4/language/irrelevance.html
Encoding of Predicate Subtyping and Proof Irrelevance in the $\lambda\Pi$-calculus Modulo Theory

\[
\begin{align*}
[x]_{\Gamma} &= x \\
[Prop]_{\Gamma} &= \text{prop} \\
[Type]_{\Gamma} &= \text{type} \\
[M \cdot N]_{\Gamma} &= [M]_{\Gamma} \cdot [N]_{\Gamma} \\
[\lambda x : T, M]_{\Gamma} &= \lambda x : [T]_{\Gamma}, [M]_{\Gamma} \\
[(x : T) \rightarrow U]_{\Gamma} &= [T]_{\Gamma} \rightarrow (\lambda x : [T]_{\Gamma}, [U]_{\Gamma} \cdot x : T) \\
[(x : T) \rightarrow P]_{\Gamma} &= \forall [T]_{\Gamma} \cdot (\lambda x : [T]_{\Gamma}, [P]_{\Gamma} \cdot x : T) \\
[(h : P) \rightarrow Q]_{\Gamma} &= [P]_{\Gamma} \Rightarrow (\lambda h : [P]_{\Gamma}, [Q]_{\Gamma} \cdot h : P) \\
\end{align*}
\]

when $\Gamma \vdash \text{PVS} \cdot T : \text{Type}$ and $\Gamma, x : T \vdash \text{PVS} \cdot P : \text{Prop}$.

**Figure 7** Translation from $\lambda\text{HOL}$ to $\lambda\Pi/\equiv$.

\[
\begin{align*}
\text{psub} \cdot T \cdot P \cdot M \cdot N &= \text{psub} \cdot [T]_{\Gamma} \cdot [P]_{\Gamma} \cdot [M]_{\Gamma} \\
\text{pair} \cdot T \cdot P \cdot M \cdot N &= \text{pair} \cdot [T]_{\Gamma} \cdot [P]_{\Gamma} \cdot [M]_{\Gamma} \cdot [N]_{\Gamma} \\
\text{fst} \cdot T \cdot P \cdot M &= \text{fst} \cdot [T]_{\Gamma} \cdot [P]_{\Gamma} \cdot [M]_{\Gamma} \\
\text{snd} \cdot T \cdot P \cdot M &= \text{snd} \cdot [T]_{\Gamma} \cdot [P]_{\Gamma} \cdot [M]_{\Gamma} \\
\end{align*}
\]

**Figure 8** Translation from PVS-Cert to $\lambda\Pi/\equiv$.

$T, U$ designate terms of type $\text{Type}$; $M, N, t, u$ designate expressions that have a type $T : \text{Type}$; $P, Q$ are propositions of type $\text{Prop}$, or predicates of type $T \rightarrow \text{Prop}$; $h$ stands for a proof typed by a proposition.

Typing judgements in PVS-Cert are noted with $\vdash_{\text{PVS}}$, and typing judgements in $\lambda\Pi/\equiv$ are noted with $\vdash_{\lambda\Pi/\equiv}$.

**Lemma 8** (Preservation of substitution). If $\Gamma, x : U, \Delta \vdash_{\text{PVS}} M : T$ and $\Gamma \vdash_{\text{PVS}} N : T$, then $\Gamma, \{x \mapsto N\} \cdot M \equiv_{\lambda\Pi/\equiv} [M]_{\Gamma, x \mapsto U, \Delta} \cdot [N]_{\Gamma}$.

**Proof.** By structural induction on $M$. ◀

**Lemma 9** (Preservation of equivalence). Let $M$ and $N$ be two well typed terms in $\Gamma$.

1. If $M \leftrightarrow_{\text{PVS}} N$, then $[M]_{\Gamma} \equiv_{\lambda\Pi/\equiv} [N]_{\Gamma}$.
2. If $M \equiv_{\text{PVS}} N$, then $[M]_{\Gamma} \equiv_{\lambda\Pi/\equiv} [N]_{\Gamma}$.

\[
\begin{align*}
[[T]]_{\Gamma} &= \text{El} \cdot [T]_{\Gamma} \quad \text{when } \Gamma \vdash_{\text{PVS}} T : \text{Type} \\
[[T]]_{\Gamma} &= \text{Prf} \cdot [T]_{\Gamma} \quad \text{when } \Gamma \vdash_{\text{PVS}} T : \text{Prop} \\
[[\text{Kind}]] &= \text{Kind} \\
[[\text{Type}]] &= \text{Type}
\end{align*}
\]

**Figure 9** Translation of types from PVS-Cert to $\lambda\Pi/\equiv$. 

symbol stack : Type  symbol empty : El stack  symbol t : Type

definition nonempty_stack?(s : El stack) := s ≠ empty
definition nonempty_stack := psub nonempty_stack?
symbol push : El stack → El t → El nonempty_stack

symbol pop : El nonempty_stack → El stack

symbol pop_push(𝑥 ∶ El 𝑡)(𝑠 ∶ El stack) : Prf(pop(push 𝑥 𝑠) = 𝑠)

theorem pop2push2(𝑥 𝑦 ∶ El 𝑡)(𝑠 ∶ El stack) : Prf(pop(pop(push 𝑥 𝑠))) = 𝑠)

Figure 10 Specification for stacks.

definition bound := ...

definition bounded := psub(𝜆𝑛, 𝑛 ≤ bound)

definition ℓ_1 := bcons(pair zero 𝑝₁) bnil

definition ℓ_2 := bcons(pair zero 𝑝₂) bnil

Figure 11 Specification of sorted lists.

Figure 12 Definition of two sorted lists with different proofs.
Proof. Each item is proved separately.
1. Taking back the notations of the proof of Lemma 3, we show that
a. computational steps of $\leftrightarrow_{\beta_{\text{fast}}}$ are preserved,
b. equational steps of $\leftrightarrow_{\beta_{\text{pi}}}$ are preserved.
These two properties are shown by induction on a context $C$ such that $M = C[M] R C[N] = N$ where $R$ is any of the two relations and the $R$ relation is applied at the head of $M$ and $N$.

We will only detail the initialisation of inductions, the other cases being straightforward.

Preservation of Computation There are two possible cases,

Case $M = ((\lambda x, t) u) \leftrightarrow_{\beta} \{ x \mapsto u \} t$, we have,

$$\Gamma \vdash \beta \{ x \mapsto u \} t$$

where the equivalence is given by Lemma 8.

Case $M = \text{fst} T_1 P_1 (\text{pair} T_0 P_0 t h) \leftrightarrow_{\text{fst}} t$, we have the following equalities

$$\Gamma \vdash \text{fst} T_1 P_1 \ \text{pair} T_0 P_0 t h = \text{fst} [T_1]_\Gamma \ \text{pair} [T_0]_\Gamma \ [P_0]_\Gamma \ [t]_\Gamma \ [h]_\Gamma \equiv [\text{pair} T_0 P_0 t h]_\Gamma$$

with the last equivalence provided by Equation (6).

Preservation of Proof Irrelevance Assume that $M = \text{pair} T P t h \leftrightarrow_{\text{pi}} \text{pair} T P t h'$

$$\Gamma \vdash \text{pair} T P t h = \text{pair} [T]_\Gamma \ [P]_\Gamma \ [t]_\Gamma \ [h]_\Gamma \equiv \text{pair} [T P t h]_\Gamma$$

where the equivalence is given by Equation (5).

2. By Lemma 3, we know that there are $H_0$ and $H_1$ such that $M(\leftrightarrow_{\beta_{\text{fast}}})^* H_0 (\leftrightarrow_{\beta_{\text{pi}}})^* H_1 (\leftrightarrow_{\beta_{\text{fast}}})^* N$. For $R \in (\leftrightarrow_{\beta_{\text{pi}}}, \leftrightarrow_{\beta_{\text{fast}}})$, we have $t(R^g)^* u \Rightarrow [t] \equiv [u]$ by induction on the number of $R^g$ steps, using Item 1 for the base case. Therefore, $[M]_\Gamma \equiv [H_0]_\Gamma \equiv [H_1]_\Gamma \equiv [N]_\Gamma$, which gives, by transitivity of $\equiv_{\text{pi}}$, $[M]_\Gamma \equiv [N]_\Gamma$.

\[\Box\]

Theorem 10 (Correctness). If $\Gamma \vdash_{\text{VS}} M : T$, then $[\Gamma]_{\lambda_{\text{VS}}} [M]_\Gamma : [T]_\Gamma$. For all $\Gamma$, if $\Gamma \vdash_{\text{VS}} W F$, then $[\Gamma]_{\lambda_{\text{VS}}} W F$.

Proof. By induction on the typing derivation of $\Gamma \vdash_{\text{VS}} M : T$ and case distinction on the last inference rule.

empty \[\emptyset \vdash_{\text{VS}} W F\]

We have $[\emptyset] = \emptyset$ and $\emptyset \vdash_{\lambda_{\text{VS}}} W F$.

\[\frac{\text{decl } \Gamma, v : T \vdash_{\text{VS}} s}{\Gamma \vdash_{\text{VS}} T : s \ \text{if } v \notin \Gamma}\]

We have $[\Gamma, v : T] = [\Gamma], v : [T]_\Gamma$. By induction hypothesis, we have $[\Gamma]_{\lambda_{\text{VS}}} [T]_\Gamma : [s]_\Gamma$, for $s \in S$ and hence $[s]_\Gamma$ is either Prop by conversion (because $\text{El prop} \equiv_{\lambda_{\text{VS}}} \text{Prop}$), Type or Kind. If $s$ is Kind, then $T$ is Type. Since $[\Gamma]_{\lambda_{\text{VS}}} \text{Type}$ because $\Sigma_{\text{PC}}(\text{Type}) = (\emptyset, (\text{TYPE}, \text{KIND}))$, we can derive with the declaration rule $[\Gamma, v : T] : [T]_{\lambda_{\text{VS}}} W F$ because $[T] \cdot \text{Type}$. Otherwise, $s$ is Type or Prop and $[T] = \xi [T]_\Gamma$, where $\xi$ is El or Prf. By typing of El or Prf (with the signature), $[\Gamma]_{\lambda_{\text{VS}}} [T]_\Gamma : \text{TYPE}$ and finally, $[\Gamma, v : T]_{\lambda_{\text{VS}}} W F$ by application of the declaration rule.
\[
\frac{\Gamma \vdash_{PVS} WF \quad v : T \in \Gamma}{\text{var}}
\]

By definition, \(\llbracket v \rrbracket = v\) and by induction hypothesis, \(\llbracket \Gamma \rrbracket \vdash_{\text{SN/}} \text{WF}\). Since \(v : T \in \Gamma\), by

\[
\frac{\Delta \subseteq \Gamma, \Delta \vdash_{PVS} WF \quad \text{such that,} \quad v : [T]_\Delta \in \llbracket \Gamma \rrbracket}{\text{and finally} \quad \llbracket \Gamma \rrbracket \vdash_{\text{SN/}} v : [T]_\Delta, \text{because contexts are well formed.}}
\]

\[
\frac{\Gamma \vdash_{PVS} (s_1, s_2) \in A}{\text{sort}}
\]

First, \([s_1]\) is either \text{prop} or \text{type}. In the former case, \([s_2]\) = \text{Type} and because \(\llbracket \Gamma \rrbracket \vdash_{\text{SN/}} \text{WF}\)

(by induction hypothesis) and \(\Sigma^{PC}(\text{prop}) = (\emptyset, (\text{Type, TYPE}))\), we have \(\llbracket \Gamma \rrbracket \vdash_{\text{SN/}} \text{prop : Type}\). The same procedure holds for \(s_1 = \text{Type}\) and \(s_2 = \text{Kind}\).

\[
\frac{\Gamma \vdash_{PVS} T : s_1 \quad \Gamma, x : T \vdash_{PVS} U : s_2 \quad (s_1, s_2, s_3) \in \mathcal{P}}{\text{prod}}
\]

We only detail for the product \((\text{Type, Prop, Prop})\), others being processed similarly. We have \(\llbracket (x : T) \rightarrow U \rrbracket_\Gamma = \forall [T]_\Gamma \left( \lambda x : [T]_\Gamma : \llbracket U \rrbracket_\Gamma, x : T \vdash \llbracket U \rrbracket_\Gamma \right)\). By induction hypothesis, \(\llbracket \Gamma \rrbracket \vdash_{\text{SN/}} [T] : \text{Type}\), and thus \(\llbracket \Gamma \rrbracket \vdash_{\text{SN/}} [T] : \text{Type}\) by definition. By induction hypothesis,

\[
\frac{\llbracket x : T \rrbracket \vdash_{\text{SN/}} [U] : \llbracket \text{Prop} \rrbracket, \text{and thus} \quad \llbracket \Gamma \rrbracket, x : T \vdash \llbracket U \rrbracket_\Gamma : \text{Prop by definition of} \llbracket \rrbracket \text{ and conversion which yields} \quad \llbracket \Gamma \rrbracket \vdash_{\text{SN/}} \lambda x : [T]_\Gamma : \llbracket U \rrbracket_\Gamma \rightarrow \text{Prop}}{\text{To finish, we obtain}}
\]

\[
\frac{\llbracket \Gamma \rrbracket \vdash_{\text{SN/}} \lambda x : [T]_\Gamma : \llbracket U \rrbracket_\Gamma \rightarrow \text{Prop}}{\text{by conversion. Using the typing signature} \Sigma^{PC}, \llbracket \Gamma \rrbracket \vdash_{\text{SN/}} \forall [T]_\Gamma \left( \lambda x, [T]_\Gamma, [U]_\Gamma, x : T \right) : \llbracket \text{Prop} \rrbracket \text{ which becomes, by conversion} \quad \text{Prop} \equiv \text{Elprop and definition of} \llbracket \rrbracket_\Gamma \text{, Elprop} = \llbracket \text{Prop} \rrbracket, \text{ hence,} \quad \llbracket \Gamma \rrbracket \vdash_{\text{SN/}}}
\]

\[
\frac{\forall [T]_\Gamma \left( \lambda x, [T]_\Gamma, [U]_\Gamma, x : T \right) : \llbracket \text{Prop} \rrbracket}{\text{∀ [T]_\Gamma (λx, [T]_\Gamma, [U]_\Gamma, x, T) : [Prop]}}
\]

\[
\frac{\Gamma, v : T \vdash \text{M} : U \quad \Gamma \vdash_{PVS} (v : T) \rightarrow U : s}{\text{app}}
\]

We have \(\llbracket \lambda v : T, M \rrbracket_\Gamma = \lambda v : [T]_\Gamma : \llbracket M \rrbracket_\Gamma, \text{by induction hypothesis,} \quad \llbracket \Gamma, v : T \rrbracket \vdash_{\text{SN/}} \llbracket \text{M} \rrbracket_{\Gamma, v : T}\)

and by definition of \llbracket \rrbracket_\Gamma. Applying the abstraction rule in \(\Pi/\equiv\), we obtain \(\llbracket \Gamma \rrbracket \vdash_{\text{SN/}} \lambda v : [T]_\Gamma : \llbracket M \rrbracket_{\Gamma, v : T} : \llbracket v : [T]_\Gamma \rrbracket \rightarrow \llbracket [U]_\Gamma \rrbracket_{\Gamma, v : T}\) (with the product well typed in \(\Pi/\equiv\), since \([U]\) and \([T]\) are both of type \(\text{TYPE}\) and thus the product is of type \(\text{TYPE}\) as well).

Finally, we proceed by case distinction on sorts \(s_T, s_U\) such that \(\Gamma \vdash_{PVS} T : s_T\) and

\[
\frac{\Gamma \vdash_{PVS} U : s_U}{\text{app}}
\]

We will detail the case \((s_T, s_U) = (\text{Type, Prop})\). We have \((v : [T]_\Gamma) \rightarrow \llbracket [U]_\Gamma, v : T \equiv \text{Prf}((\forall [T]_\Gamma (\lambda x, [T]_\Gamma, [U]_\Gamma, x : T)) = \llbracket (v : T) \rightarrow [U]_\Gamma \rrbracket\) which allows to conclude.

\[
\frac{\Gamma \vdash_{PVS} M : (v : T) \rightarrow U \quad \Gamma \vdash_{PVS} N : T}{\text{app}}
\]

By induction hypothesis and conversion, we have \(\llbracket \Gamma \rrbracket \vdash_{\text{SN/}} \llbracket \text{M} \rrbracket_\Gamma : \llbracket v : [T]_\Gamma \rrbracket \rightarrow \llbracket [U]_\Gamma, v : T\)

(by case distinction on the sorts of \(T\) and \(U\)) and \(\llbracket \Gamma \rrbracket \vdash_{\text{SN/}} \llbracket \text{N} \rrbracket_\Gamma : \llbracket T \rrbracket_\Gamma\). Since \([M N]_\Gamma = \llbracket M \rrbracket \llbracket N \rrbracket_\Gamma\), we obtain using the application rule \(\llbracket \Gamma \rrbracket \vdash_{\text{SN/}} \llbracket \text{M N} \rrbracket_\Gamma : \llbracket v \mapsto [N]_\Gamma \rrbracket \llbracket \text{U} \rrbracket_{\Gamma, v : T}\) and by

Lemma 8, we obtain \(\llbracket \Gamma \rrbracket \vdash_{\text{SN/}} \llbracket \text{M N} \rrbracket : \llbracket \{v \mapsto [N]_\Gamma \} \llbracket \text{U} \rrbracket_{\Gamma, v : T}\).

\[
\frac{\Gamma \vdash_{PVS} M : U \quad \Gamma \vdash_{PVS} T : s \quad T \equiv_{PVS} U}{\text{app}}
\]

By hypothesis, there is a type \(U\) such that \(\Gamma \vdash_{PVS} M : U\), and \(T \equiv U\), and there is a sort \(s\) such that \(\Gamma \vdash_{PVS} T : s\). By induction hypothesis, \(\llbracket \Gamma \rrbracket \vdash_{\text{SN/}} \llbracket \text{M} \rrbracket_\Gamma = \llbracket U \rrbracket_\Gamma\).

We now prove that if \(T \equiv U\), then \(\llbracket T \rrbracket_\Gamma = \llbracket U \rrbracket_\Gamma\) and \(\Gamma \vdash_{\text{SN/}} \llbracket \text{U} \rrbracket : \text{TYPE}\). It will allow us to conclude using the conversion rule in \(\Pi/\equiv\).

By Lemma 3, we have \(T \equiv_{\beta\text{f}} T' \equiv_{\beta\text{f}} U' \equiv_{\beta\text{f}} U\) and \(T(\equiv_{\gamma\text{f}} U' \equiv_{\gamma\text{f}} U) = T(\equiv_{\beta\text{f}}) U'(\equiv_{\beta\text{f}}) U\).

Because \(\equiv_{\beta\text{f}}\) preserves typing (Lemma 3), we have \(\Gamma \vdash_{PVS} U' : s\). By [7, Lemma 43],
\[ \Sigma(f) = (\vec{x}, T, U, s) \]

\[ \text{encoding of Predicate Subtyping and Proof Irrelevance in the } \lambda \Pi \text{-calculus Modulo Theory} \]

\[ \Gamma \vdash_{\text{PVS}} T : s. \] By Lemma 9, \([\Gamma]_{\lambda \Pi} \equiv [U]_{\Gamma}\).

If \(s = \text{Prop}\), then \([\Gamma]_{\lambda \Pi} = \text{Prf}[\Gamma]_{\lambda \Pi} \equiv \text{Prf}[U]_{\Gamma} = [U]_{\Gamma}\). Moreover we have \([\Gamma]_{\lambda \Pi} \equiv [T]_{\Gamma} : \text{TYPE}\) because, by induction hypothesis, \([\Gamma]_{\lambda \Pi} : \text{Prf}[\text{Prop}] = \text{El}[\text{Prop}] = \text{El}[U]_{\Gamma} = [U]_{\Gamma}\). 

\((p : \text{Prop} \vdash_{\SigmaPC} \text{Prf} : \text{TYPE} : \text{KIND})\). If \(s = \text{Type}\), \([\Gamma]_{\lambda \Pi} = \text{El}[\text{Type}]_{\Gamma} \equiv \text{El}[U]_{\Gamma} = [U]_{\Gamma}\). By induction hypothesis, \([\Gamma]_{\lambda \Pi} : [\text{Type}]_{\Gamma} = \text{Type}\). If \(s = \text{Kind}\), then \(T = U = \text{Type}\) (Type is the only inhabitant of Kind). Finally, \([\text{Type}]_{\Gamma} = \text{Type} : \text{TYPE}\).

\[ \vec{x} : \vec{T} : U : \Gamma \vdash t_{i} : \{ (x_{j} \mapsto t_{j})_{j < i} \} T_{i} \]

\[ \Sigma(f) = (\vec{x}, \vec{T}, U, s) \]

We first observe from Figure 6 that for each \(f \in \Sigma_{\text{PVS}}\), we have a counterpart symbol \(\hat{f} \in \Sigma_{\text{PC}}\) such that if \(\Sigma_{\text{PVS}}(f) = (\vec{x}, \vec{T}, U, s)\), then \(\Sigma_{\text{PC}}(\hat{f}) = (\vec{x}, [\vec{T}], [U]_{\vec{T}}, \text{TYPE})\).

By induction hypothesis, for each \(i\), we have \([\Gamma]_{\lambda \Pi / \equiv} t_{i} : \{ (x_{j} \mapsto t_{j})_{j < i} \} T_{i}\) which we can write as, thanks to Lemma 8, \([\Gamma]_{\lambda \Pi / \equiv} t_{i} : \{ (x_{j} \mapsto t_{j})_{j < i} \} [T]_{i}\).

Now, using the signature rule, we are able to conclude \([\Gamma]_{\lambda \Pi / \equiv} \hat{f}[t_{i}]_{\Gamma} : \{ (x \mapsto \hat{t}) \} [U]_{\Gamma}\). By Lemma 8, we obtain \([\Gamma]_{\lambda \Pi / \equiv} \hat{f}[t_{i}]_{\Gamma} : \{ (x \mapsto \hat{t}) \} [U]_{\Gamma}\). Moreover, we have taken care to define the translation in Figure 8 such that \([f(\vec{t})] = \hat{f}[\hat{t}]\). □

### 5  Mechanised Type Checking

The encoding of PVS-CERT into \(\lambda \Pi / \equiv\) can be used to proof check terms of PVS-CERT using a type checker for \(\lambda \Pi / \equiv\). But because of the rule

\[ \Gamma \vdash t : B \quad \Gamma \vdash A : s \quad A \equiv B \]

\[ \Gamma \vdash t : A \quad (\lambda \Pi / \equiv\text{-conv}) \]

type checking is decidable only if \(\equiv\) so is. A decidable relation equivalent to \(\equiv\) can be obtained using the convertibility relation stemming from the rewriting relation of a convergent rewrite system, yielding the type system \(\lambda \Pi / \equiv(R\text{ for rewriting})\). Consequently, while type checkers cannot be provided for \(\lambda \Pi / \equiv\) in general, they can be for \(\lambda \Pi / R\), as can be seen with Dedukti\(^3\).

Such rewrite systems can be obtained through completion procedures [5]. However, completion procedures rely on a well-founded order that cannot be provided here because of Equation (5) which cannot be oriented since each side of the equation has a free variable which is not in the other side.

A possible solution would be to rewrite all proofs of a pair to a canonical proof with a rule of the form

\[ \text{pair } tp m h \leftrightarrow \text{pair } tp m (\text{canon} t pm) \]

where \(t : \text{Type}, p : \text{El} t \rightarrow \text{Prop}, m : \text{El} t \vdash \text{canon} t pm : \text{Prf}(pm) : \text{TYPE}\). But this creates a rewrite rule that duplicates three variables.

Otherwise, as noted in [22], the addition of a symbol to the signature can circumvent the issue. Hence, we add a symbol for proof irrelevant pairs, and make it equal to pairs

\[ t : \text{Type}, p : \text{El} t \rightarrow \text{Prop}, m : \text{El} t \vdash \text{pair}^\dagger t pm : \text{El}(\text{psub} t p) : \text{TYPE} \quad (25) \]

\[ \text{pair } tp m h = \text{pair}^\dagger t pm \quad (26) \]

\(^3\) https://github.com/Dedukti/lambdaPi.git
such that \((\text{pair } t p m h) \equiv (\text{pair}^\dagger t p m) \equiv (\text{pair } t p m h')\). The new set of identities given by Equations (6), (17) to (20) and (26) can be completed into a rewrite system \(R\) which is equivalent to the equations:

\[
\begin{align*}
(\lambda x : T, t) u & \leftrightarrow \{x \mapsto u\} t & \text{(27)} \\
\text{pair } t p m h & \leftrightarrow \text{pair}^\dagger t p m & \text{(28)} \\
\text{fst } t_0 p_0 (\text{pair}^\dagger t_1 p_1 m) & \leftrightarrow m & \text{(29)} \\
\text{El prop} & \leftrightarrow \text{Prop} & \text{(30)} \\
\text{Prf}(\forall t p) & \leftrightarrow (x : \text{El } t) \rightarrow \text{Prf}(p x) & \text{(31)} \\
\text{El}(t \Rightarrow u) & \leftrightarrow (x : \text{El } t) \rightarrow \text{El}(u x) & \text{(32)} \\
\text{Prf}(p \Rightarrow q) & \leftrightarrow (h : \text{Prf } p) \rightarrow (\text{Prf } q h) & \text{(33)}
\end{align*}
\]

\[\text{Figure 13} \quad \text{Rewrite system } R \text{ resulting from the completion of the equations of the encoding of PVS-Cert in } \lambda \Pi/\equiv.\]

\[\text{Proposition 11.} \quad \text{Let } \leftrightarrow_R \text{ be the closure by context and substitution of the rewrite rules of Figure 13, and } \equiv_R \text{ be the smallest equivalence containing } \leftrightarrow_R. \text{ Then, for all } M, N \in \mathcal{T}(\Sigma^\text{PC}, S^\Pi, \mathcal{V}), \text{ if } M \equiv_N \text{ then } M \equiv_R N.\]

\[\text{Proof.} \quad \text{It suffices to prove that every equation of PVS-Cert is included in } \equiv_R. \text{ This is immediate for the Equations (17) to (20) and } (\beta) \text{ since they are equal to the rules (27) and (30) to (33). For the Equation (5), we have } \text{pair } t p m h \leftrightarrow_R \text{pair}^\dagger t p m \leftrightarrow_R \text{pair } t p m h_1. \text{ Finally, for the Equation (6), we have } \text{fst } t_0 p_0 (\text{pair}^\dagger t_1 p_1 m) \leftrightarrow_R \text{fst } t_0 p_0 (\text{pair}^\dagger t_1 p_1 m) \leftrightarrow_R m. \]

\[\text{Remark 12.} \quad \text{Rewrite system } R \text{ is confluent because orthogonal.}\]

\[\text{Conclusion} \quad \text{This work provides an encoding of predicate subtyping with proof irrelevance into the } \lambda \Pi\text{-calculus modulo theory, } \lambda \Pi/\equiv [3]. \text{ We first recall PVS-Cert, an extension of higher-order logic with predicate subtyping and proof irrelevance [16]. We then provide a } \lambda \Pi/\equiv \text{ signature to encode terms of PVS-Cert, and prove that the encoding is correct: if a PVS-Cert type is inhabited, then its translation in } \lambda \Pi/\equiv \text{ is inhabited too. Finally, we show that the equational theory of our encoding is equivalent to a confluent set of rewrite rules which enable us to use Dedukti to type check encoded specifications.}\]

\[\text{However, two important problems are left open. First, is our encoding complete, that is, is a PVS-Cert type inhabited if its translation so is? Second, is the confluent rewrite system used in the encoding terminating? We believe that these two properties hold but leave their study for future work.}\]
The encoding of PVS-Cert in \(\lambda\Pi/R\) is the stepping stone towards an automatic translator from PVS to Dedukti. Indeed, PVS does not have proof terms in its syntax, and consequently type checking is undecidable. The creation of PVS-Cert allows to convert PVS terms to a syntax whose type checking is decidable. This was the work of F. Gilbert in [16]. Now we are able to express this decidable syntax in \(\lambda\Pi/R\) and hence in Dedukti. However, the type system proposed here only allows to coerce from a type to its direct supertype or a subtype, that is, we can go from \((\text{psub}(\text{psub} P) Q) Q\) to \(\text{psub} P\) in one coercion, but we cannot coerce from \((\text{psub}(\text{psub} P) Q) Q\) to \(i\), whereas PVS can. Consequently, an algorithm to elaborate the correct sequence of coercions is needed to obtain terms that can be type checked in Dedukti.

Other features of PVS can be integrated into PVS-Cert and the encoding: dependent types like \((\text{psub list}(\lambda \ell, \text{length } \ell = \ell) = \ell))\), recursive definitions of functions, and dependent records. With those features encoded, almost all the standard library of PVS can be translated to Dedukti.

Finally, while the previous points were concerned with the translation of specifications from PVS, we may also want to translate proofs developed in PVS. These proofs are witnesses of type correctness conditions (TCC), which are required to type check terms. Since PVS is a highly automated prover, proof terms often become application of complex tactics that cannot be mimicked into Dedukti. However, proof terms may either be provided by hand, emulating the interaction provided by TCC’s, or we may call external solvers [18].

Acknowledgements. The authors thank Gilles Dowek very much for his remarks on previous versions of this work.

References

4 http://www.cs.rug.nl/~grl/ar06/prelude.html


Frederic Gilbert. Extending higher-order logic with predicate subtyping: application to PVS. Theses, Université Sorbonne Paris Cité, April 2018. URL: https://tel.archives-ouvertes.fr/tel-02058937.

Mohamed Il Haddad, Guillaume Burel, and Frédéric Blanqui. EKSTRAKTO A tool to reconstruct deducted proofs from TSTP files (extended abstract). In Giselle Reis and Haniel Barbosa, editors, *Proceedings Sixth Workshop on Proof eXchange for Theorem Proving, PxTP 2019, Natal, Brazil, August 26, 2019*, volume 301 of *EPTCS*, pages 27–35, 2019. URL: https://doi.org/10.4204/EPTCS.301.5, doi:10.4204/EPTCS.301.5.


Encoding of Predicate Subtyping and Proof Irrelevance in the \(\lambda\Pi\)-calculus Modulo Theory


