A Framework for Proof-carrying Logical Transformations

Quentin Garchery
Université Paris-Saclay, CNRS, Inria, LMF, 91405, Orsay, France

In various provers and deductive verification tools, logical transformations are used extensively in order to reduce a proof task into a number of simpler tasks. Logical transformations are often part of the trusted base of such tools. In this paper, we develop a framework to improve confidence in their results. We follow a modular and skeptical approach: transformations are instrumented independently of each other and produce certificates that are checked by a third-party tool. Logical transformations are considered in a higher-order logic, with type polymorphism and built-in theories such as equality and integer arithmetic. We develop a language of proof certificates for them and use it to implement the full chain of certificate generation and certificate verification.

1 Introduction

General Context and Motivation. Verifying a program is meant to improve its soundness guarantees and relies on the trust towards the verification tool. Given how difficult it can be to verify relatively simple programs, most tools try to simplify this process and to make it as automatized as possible, which can drastically extend their trusted code base.

Consider deductive program verification, where the program to verify is annotated, and, in particular, given a specification. In this setting, the code is analyzed against its specification thus generating *proof tasks*, logical statements upon which depends the program correctness. To discharge a proof task, one can first apply a *logical transformation* which reduces it to a number of new proof tasks which are hopefully easier to discharge. Transformations are powerful tools that can, for example, be applied to translate a task into a prover's logic before calling it.

The main objective of this article is to improve trust in those logical transformations. This work is decisive because logical transformations are general and can be used in many different settings. We apply our method to the deductive program verification tool Why3 [6], which makes extensive use of logical transformations. In fact, they are at the core of its interactive theorem proving feature and are necessary to be able to encode proof tasks into the logic of one of the dozens of third-party automatic theorem provers available inside Why3. The implementation of the transformations adds up to a total of more than 17 000 lines of OCaml code. This code being in the trusted Why3 code base, it represents an interesting case study.

Example 1. *Suppose that we have a proof task where we have to prove that* $p\ (y * y)$ *holds for any integer* y *of the form* $y = 2 * x + 1$ *and any integer predicate* p *that satisfies the hypothesis* H *stating that* $\forall i : int.\ p\ (4 * i + 1)$. *Finding how to instantiate the hypothesis* H *is difficult or even impossible for some automatic theorem provers so we cannot automatically discharge this task. The transformation* `instantiate`*, defined in Why3, simply instantiates an hypothesis and when called with arguments* H *and* $x * x + x$ *on the given task, produces the same task but with the added hypothesis that states that* $p\ (4 * (x * x + x) + 1)$. *Provers won't have to instantiate the hypothesis* H *to discharge this new task. In fact, this task can now be discharged by theorem provers capable of handling arithmetic goals on condition of translating it into the logic of the prover in question. This translation is also being done*

Chantal Keller and Mathias Fleury (Eds.): Seventh Workshop
on Proof eXchange for Theorem Proving (PxTP)
EPTCS 336, 2021, pp. 5–23, doi:10.4204/EPTCS.336.2

with the help of transformations. The entire process described in this example is to be trusted to ensure correctness of the initial program.

Contributions and Overview. In this article, we describe a practical framework to validate logical transformations. In order to define what it means for a transformation to be correct, the logical setting of proof tasks is detailed in Section 2. We follow a skeptical approach [4]: certificates, defined in Section 3, are generated every time a transformation is applied and are checked independently at a later time. Contrary to the autarkic approach, used for example for some automatic theorem provers [28], which would consist here in verifying directly the transformations, the skeptical approach has the benefit of not fixing the implementation of the transformations. Our work is based on certificates with holes, a notion that is, to our knowledge, new in the setting of the skeptical approach. This allows for modular development, where certificates can be built incrementally and transformations can be composed and defined independently. We extend our framework with some key interpreted theories in Section 4 and show how to do so for any other interpreted theory along the way. The checkers for our certificates can also be defined independently, as it is done in Section 5. In fact, we designed two checkers and one of them is based on Lambdapi/Dedukti [3], an off-the-shelf proof assistant. This has also led us to develop a translation procedure for proof tasks into the $\lambda\Pi$-Calculus modulo rewriting. This approach, while applicable to logical transformations in general, has been applied to the program verification tool Why3 for a number of its transformations including transformations dealing specifically with higher-order logic. We conclude this article by evaluating this application to Why3 in Section 6. The source code for the whole work described in this article is available in the Why3 repository [21].

2 Logical Setting

We present the logical setting used throughout this article. The goal here is to define logical transformations and the proof tasks they are applied to.

2.1 Types and Terms

Proof tasks are formed from typed terms and those terms are meant to designate both the terms from the program and the formulas stating properties about them. We use the Hindley-Milner type system [29] except that our terms are explicitly quantified over types. Names are taken from an infinite set of available identifiers which is designated by *ident*.

Types are described by a *type signature* I, a set of pairs of the form '$\iota : n$' composed of an *ident* called type symbol and an integer representing its arity. Sets are denoted by separating their elements with commas. Note that according to the following grammar, type symbols are always completely applied.

$$\begin{array}{rcll} \mathit{type} & ::= & \alpha & \text{type variable} \\ & | & \mathit{prop} & \text{type of formulas} \\ & | & \mathit{type} \rightsquigarrow \mathit{type} & \text{arrow type} \\ & | & \iota(\mathit{type}, \ldots, \mathit{type}) & \text{type symbol application} \end{array}$$

Terms have polymorphic types and new terms can be built by quantifying over terms of any type. Quantification over type variables is explicit and restricted to only be in the prenex form. Note that the application uses the Curry notation, i.e., a function term is applied to a single argument term at a time. The application is left-associative and the type arrow $\rightsquigarrow$ is right-associative.

$$
\begin{array}{rcll}
term_{poly} & ::= & term_{mono} & \\
 & | & \Pi\alpha.\ term_{poly} & \text{type quantifier} \\
 & & & \\
term_{mono} & ::= & x & \text{variable} \\
 & | & \top & \text{true formula} \\
 & | & \bot & \text{false formula} \\
 & | & \neg\ term_{mono} & \text{negation} \\
 & | & term_{mono}\ op\ term_{mono} & \text{logical binary operator} \\
 & | & term_{mono}\ term_{mono} & \text{application} \\
 & | & \lambda x : type.\ term_{mono} & \text{anonymous function} \\
 & | & \exists x : type.\ term_{mono} & \text{existential quantifier} \\
 & | & \forall x : type.\ term_{mono} & \text{universal quantifier} \\
 & & & \\
op & ::= & \wedge & \text{conjonction} \\
 & | & \vee & \text{disjunction} \\
 & | & \Rightarrow & \text{implication} \\
 & | & \Leftrightarrow & \text{equivalence}
\end{array}
$$

The *(term) substitution* of variable x by term u in term t is written $t[x \mapsto u]$ and $t[\alpha \mapsto \tau]$ is the *(type) substitution* of type variable α by type τ in term t. A *signature* Σ is a set of pairs of the form '$x : \tau$' composed of a variable and its type; this type should be understood as quantified over all of its type variables.

Definition 2 (Typing)**.** *We write $I \mid \Sigma \Vdash t : \tau$ when the term t has no free type variables and has type τ in type signature I and signature Σ. We omit I when it is clear from the context.*

The complete set of rules defining this predicate is given in Appendix A. Remark that, in the case where t is an element of $term_{mono}$ then the predicate $\Sigma \Vdash t : \tau$ implies that t has no type variables: t is monomorphic.

2.2 Proof Tasks

Proof tasks represent sequents in higher-order logic, they are formed from two sets of premises: a set of hypotheses and a set of goals. A *premise* is a pair of the form '$P : t$' composed of an *ident* and a $term_{poly}$ representing a formula.

Definition 3 (Proof Task)**.** *Let I be a type signature, Σ be a signature, Γ and Δ be sets of premises.* Proof tasks *are denoted by $I \mid \Sigma \mid \Gamma \vdash \Delta$ which represents the sequent where goals, given by Δ, and hypotheses, given by Γ, are written in the signature Σ with types in I. We allow ourselves to omit I and, possibly, Σ, when they are clear from the context.*

A task $T := I \mid \Sigma \mid \Gamma \vdash \Delta$ is said to be *well-typed* when every premise $P : t$ from Γ or Δ is such that $I \mid \Sigma \Vdash t : prop$. The *validity* of a task is only defined when it is well-typed. In this case, the task T is said to be valid when every model of I, Σ and every formula in Γ is also a model of some formula in Δ.

Example 4. *Consider the task* $I \mid \Sigma \mid \Gamma \vdash \Delta$ *with*

$$
\begin{aligned}
I :=\ & color : 0,\ set : 1 \\
\Sigma :=\ & red : color(),\ green : color(),\ blue : color(), \\
& empty : set(\alpha),\ add : \alpha \rightsquigarrow set(\alpha) \rightsquigarrow set(\alpha), \\
& mem : \alpha \rightsquigarrow set(\alpha) \rightsquigarrow prop \\
\Gamma :=\ & H_1 : \Pi\alpha.\ \forall x : \alpha.\ \forall y : \alpha.\ \forall s : set(\alpha).\ mem\ x\ s \Rightarrow mem\ x\ (add\ y\ s), \\
& H_2 : \Pi\alpha.\ \forall x : \alpha.\ \forall s : set(\alpha).\ mem\ x\ (add\ x\ s) \\
\Delta :=\ & G : mem\ green\ (add\ red\ (add\ green\ empty))
\end{aligned}
$$

This task defines the types color and set with associated symbols red, green, blue, empty, add and mem. The type symbol declaration $set : 1$ *defines a type symbol set of arity* 1 *for polymorphic sets. The signature declaration* $add : \alpha \rightsquigarrow set(\alpha) \rightsquigarrow set(\alpha)$ *allows us to declare a function that can be used to add an element of any type to a set containing elements of the same type. This task also defines hypotheses such that the predicate mem holds if its first argument is contained in the second argument. For instance, the hypothesis* H_2 *is applicable to sets of any type, and states that every set contains the element that has just been added to it. With the given goal, this task is valid.*

2.3 Logical Transformations

A *logical transformation* is a function that takes a task as input and returns a list of tasks. Lists are denoted by separating their elements with semicolons. We say that a transformation is applied on an *initial task* and returns *resulting tasks*. A transformation can fail, in this case the whole process is terminated and we do not have to prove the correctness of the application. If a transformation succeeds, we want the verification to be based on the validity of the resulting tasks and to be able to forget about the initial task. This is why we say that a *transformation application is correct* when the validity of each resulting task implies the validity of the initial task. In Example 1, the transformation `instantiate` returns the initial task modified by adding the instantiated hypothesis to it. When a transformation application is correct, it only remains to prove that this resulting task is valid in order to make sure that the initial task is also valid. This is our approach: we certify transformation applications, thus relating initial and resulting tasks.

3 Certificates

To verify a transformation, we instrument it to produce a certificate and check each application of the transformation thanks to the corresponding certificate. We first define our own certificate format with the goal of making the verification of those certificates as easy as possible. We show how to improve modularity and ease of use in a second time in Section 5.1.

3.1 Syntax

An excerpt of the recursive definition of certificates is given in Figure 1. More certificates will be detailed on their own in Section 4 and the others won't be presented in this article for brevity. We call these certificates the *kernel certificates*. To make certificates easier to check, we design them in such a way that they are very precise. For example, the Boolean values indicate whether the premise is an hypothesis

$$
\begin{array}{rcl}
\mathit{cert} & ::= & \mathsf{KHole}(\mathit{task}) \\
& | & \mathsf{KTrivial}(\mathit{bool}, \mathit{ident}) \\
& | & \mathsf{KAssert}(\mathit{ident}, \mathit{term}_{\mathit{poly}}, \mathit{cert}, \mathit{cert}) \\
& | & \mathsf{KSplit}(\mathit{bool}, \mathit{term}_{\mathit{mono}}, \mathit{term}_{\mathit{mono}}, \mathit{ident}, \mathit{cert}, \mathit{cert}) \\
& | & \mathsf{KDestruct}(\mathit{bool}, \mathit{term}_{\mathit{mono}}, \mathit{term}_{\mathit{mono}}, \mathit{ident}, \mathit{ident}, \mathit{ident}, \mathit{cert}) \\
& | & \mathsf{KIntroQuant}(\mathit{bool}, \mathit{type}, \mathit{term}_{\mathit{mono}}, \mathit{ident}, \mathit{ident}, \mathit{cert}) \\
& | & \mathsf{KInstQuant}(\mathit{bool}, \mathit{type}, \mathit{term}_{\mathit{mono}}, \mathit{ident}, \mathit{ident}, \mathit{term}_{\mathit{mono}}, \mathit{cert}) \\
& | & \mathsf{KIntroType}(\mathit{term}_{\mathit{poly}}, \mathit{ident}, \mathit{ident}, \mathit{cert}) \\
& | & \mathsf{KInstType}(\mathit{term}_{\mathit{poly}}, \mathit{ident}, \mathit{ident}, \mathit{type}, \mathit{cert}) \\
& | & \ldots
\end{array}
$$

Figure 1: Definition of Kernel Certificates (excerpt)

or a goal. Moreover, the kernel certificates have voluntarily been kept as elementary as possible and this makes it easier to trust them. In particular, this approach makes it easier to check every case (about 20 of them) when proving by induction a property of correctness of kernel certificates, as it is done in paragraph 5.3.3.

The certificates can contain tasks and each KHole node carries one of those tasks. When c is a certificate, *the leaves of* c designate the list of all tasks obtained by collecting them (in the KHole nodes) when doing an in-order traversal of the certificate tree. The leaves of a certificate are meant to be, in the end, the resulting tasks of the transformation it is certifying. A certificate with holes associated to a transformation application can be checked without needing to wait for the proof of the returned tasks to fill its holes, and this is what makes our certificates original. This design choice has been guided by our will for modularity: we want to progressively certify logical transformations.

3.2 Semantics

The semantics of certificates is defined by a binary predicate $T \downarrow c$, linking the initial task T to a certificate c. Informally, the predicate $T \downarrow c$ holds if c represents a proof of the fact that the validity of the leaves of c implies the validity of T. In Figure 2, we give the rules that cover the certificates from Figure 1. In this sense, the rules are only an excerpt of the complete set of rules (given in Appendix B) defining the predicate $T \downarrow c$. Notice that some of the certificates have dual rules for the hypotheses and the goals.

The certificate KHole is used to validate transformations that have resulting tasks and can be used directly for the identity transformation. The certificate KTrivial is used to validate a transformation application that has no resulting task when the initial task contains a trivial premise. The KAssert certificate allows to introduce a cut on a polymorphic formula. Remember that the side condition implies that this formula cannot have free type variables. The certificates KSplit and KDestruct are used to validate a transformation application that first splits a premise H. Certificate KIntroQuant is used to introduce the variable of a quantified premise. The certificate KInstQuant is used to instantiate a quantified premise with a term and the side condition ensures that this term is monomorphic. The certificates KIntroType and KInstType are used to deal with type-quantified premises.

Example 5. *Let T and T_{inst} denote, respectively, the initial and the resulting task from Example 1, and let H_{inst} be the name of the new instantiated hypothesis. Suppose that symbol plus, symbol mult and type*

$$\frac{}{\Gamma \vdash \Delta \downarrow \mathsf{KHole}(\Gamma \vdash \Delta)} \qquad \frac{}{\Gamma, H : \bot \vdash \Delta \downarrow \mathsf{KTrivial}(\mathit{false}, H)} \qquad \frac{}{\Gamma \vdash \Delta, G : \top \downarrow \mathsf{KTrivial}(\mathit{true}, G)}$$

$$\frac{\Sigma \mid \Gamma \vdash \Delta, P : t \downarrow c_1 \qquad \Sigma \mid \Gamma, P : t \vdash \Delta \downarrow c_2 \qquad \Sigma \Vdash t : \mathit{prop}}{\Sigma \mid \Gamma \vdash \Delta \downarrow \mathsf{KAssert}(P, t, c_1, c_2)}$$

$$\frac{\Gamma, H : t_1 \vdash \Delta \downarrow c_1 \qquad \Gamma, H : t_2 \vdash \Delta \downarrow c_2}{\Gamma, H : t_1 \vee t_2 \vdash \Delta \downarrow \mathsf{KSplit}(\mathit{false}, t_1, t_2, H, c_1, c_2)} \qquad \frac{\Gamma \vdash \Delta, G : t_1 \downarrow c_1 \qquad \Gamma \vdash \Delta, G : t_2 \downarrow c_2}{\Gamma \vdash \Delta, G : t_1 \wedge t_2 \downarrow \mathsf{KSplit}(\mathit{true}, t_1, t_2, G, c_1, c_2)}$$

$$\frac{\Gamma, H_1 : t_1, H_2 : t_2 \vdash \Delta \downarrow c}{\Gamma, H : t_1 \wedge t_2 \vdash \Delta \downarrow \mathsf{KDestruct}(\mathit{false}, t_1, t_2, H, H_1, H_2, c)}$$

$$\frac{\Gamma \vdash \Delta, G_1 : t_1, G_2 : t_2 \downarrow c}{\Gamma \vdash \Delta, G : t_1 \vee t_2 \downarrow \mathsf{KDestruct}(\mathit{true}, t_1, t_2, G, G_1, G_2, c)}$$

$$\frac{\Sigma, y : \tau \mid \Gamma, H : t[x \mapsto y] \vdash \Delta \downarrow c \qquad y \text{ is fresh w.r.t. } \Sigma, \Gamma, \Delta, t}{\Sigma \mid \Gamma, H : \exists x : \tau.\ t \vdash \Delta \downarrow \mathsf{KIntroQuant}(\mathit{false}, \tau, \lambda x : \tau.\ t, H, y, c)}$$

$$\frac{\Sigma, y : \tau \mid \Gamma \vdash \Delta, G : t[x \mapsto y] \downarrow c \qquad y \text{ is fresh w.r.t. } \Sigma, \Gamma, \Delta, t}{\Sigma \mid \Gamma \vdash \Delta, G : \forall x : \tau.\ t \downarrow \mathsf{KIntroQuant}(\mathit{true}, \tau, \lambda x : \tau.\ t, G, y, c)}$$

$$\frac{\Sigma \mid \Gamma, H_1 : \forall x : \tau.\ t, H_2 : t[x \mapsto u] \vdash \Delta \downarrow c \qquad \Sigma \Vdash u : \tau}{\Sigma \mid \Gamma, H_1 : \forall x : \tau.\ t \vdash \Delta \downarrow \mathsf{KInstQuant}(\mathit{false}, \tau, \lambda x : \tau.\ t, H_1, H_2, u, c)}$$

$$\frac{\Sigma \mid \Gamma \vdash \Delta, G_1 : \exists x : \tau.\ t, G_2 : t[x \mapsto u] \downarrow c \qquad \Sigma \Vdash u : \tau}{\Sigma \mid \Gamma \vdash \Delta, G_1 : \exists x : \tau.\ t \downarrow \mathsf{KInstQuant}(\mathit{true}, \tau, \lambda x : \tau.\ t, G_1, G_2, u, c)}$$

$$\frac{I, \iota : 0 \mid \Sigma \mid \Gamma \vdash \Delta, G : t[\alpha \mapsto \iota] \downarrow c \qquad \iota \notin I}{I \mid \Sigma \mid \Gamma \vdash \Delta, G : \Pi\alpha.\ t \downarrow \mathsf{KIntroType}(\Pi\alpha.\ t, G, \iota, c)}$$

$$\frac{\Gamma, H_1 : \Pi\alpha.\ t, H_2 : t[\alpha \mapsto \tau] \vdash \Delta \downarrow c \qquad \tau \text{ has no type variables}}{\Gamma, H_1 : \Pi\alpha.\ t \vdash \Delta \downarrow \mathsf{KInstType}(\Pi\alpha.\ t, H_1, H_2, \tau, c)}$$

Figure 2: Certificate Rules (excerpt)

---

symbol int are defined in the signature and type signature, then we have $T \downarrow c$ with

$$c := \mathsf{KInstQuant}(\mathit{false}, \mathit{int}, \lambda i : \mathit{int}.\ p\ (\mathit{plus}\ (\mathit{mult}\ 4\ i)\ 1),\\ H, H_{\mathit{inst}}, \mathit{int}, \mathit{plus}\ (\mathit{mult}\ x\ x)\ x, \mathsf{KHole}(T_{\mathit{inst}}))$$

Proposition 6. *If T is well-typed then every task in a derivation $T \downarrow c$ is also well-typed.*

We also assume that transformations are always applied to well-typed initial tasks and produce well-typed resulting tasks (or they fail), so that every task we consider from now on is implicitly assumed to be well-typed.

Theorem 7 (Certificate Correctness)**.** *If $T \downarrow c$ then the validity of each leaf of c implies the validity of T.*

Proof. By induction on $T \downarrow c$. □

3.3 Design Choices

The certificate rules are taken from the sequent calculus LK rules with modifications for two reasons. First, we want the production of certificates to be more natural. This is why the name KSplit is well-suited for a transformation application that, from the initial task $\Gamma, H : t_1 \vee t_2 \vdash \Delta$, returns $\Gamma, H : t_1 \vdash \Delta$ and $\Gamma, H : t_2 \vdash \Delta$. Indeed, it would be confusing to say that, from this initial task, the transformation does the left introduction of the disjunction. Second, we want to be able to implement a checker of certificates following these rules. To this end, instead of asking the checkers to find the names that were chosen by the transformation, we register these names in the certificates. For example, the KIntroQuant certificate mentions the name y of the new fresh variable that is being introduced and this is reflected in the corresponding rules.

3.4 Certifying Transformations and Composition

Definition 8 (Certifying transformation)**.** *A* certifying transformation *is a transformation that, applied on an initial task, produces, on top of a list L of resulting tasks, a certificate c such that L is the leaves of c. We say that we instrumented the transformation to produce a certificate.*

Composing transformations is useful to define a transformation from simpler ones. To compose certifying transformations, one also needs to be able to substitute certificates, that is, to replace a KHole in one certificate with another certificate. This composition allows for a modular development of certifying transformations.

4 Adding Support for Interpreted Theories

For now, our formalism implicitly makes the assumption that every symbol is uninterpreted: they are taken as fresh new symbols for every task. Still, we want some symbols (such as equality or arithmetic operations) to have a fixed interpretation. Moreover, some transformations, like induction, use specific theories and we need to add certificate steps to be able to certify them.

To make sure that the interpretation is unique, we should not quantify over the interpreted symbols at the level of the tasks. Interpreted symbols are not part of the signature or type signature of tasks. This ensures that the interpretation stays the same for the initial task and for the resulting tasks and this is enough to handle transformations on tasks that contain interpreted symbols. To handle transformations that deal with the interpreted symbols and use their properties, we extend our certificate format and add rules corresponding to their properties.

4.1 Polymorphic Equality

The polymorphic equality is interpreted. To obtain the usual properties of the equality, we add the certificates:

$$\mathsf{KEqRefl}(\mathit{term}_{\mathit{mono}}, \mathit{ident})$$
$$\mathsf{KRewrite}(\mathit{bool}, \mathit{term}_{\mathit{mono}}, \mathit{term}_{\mathit{mono}}, \mathit{term}_{\mathit{mono}}, \mathit{ident}, \mathit{ident}, \mathit{cert})$$

and three kernel rules:

$$\frac{}{\Sigma \mid \Gamma \vdash \Delta, G : x = x \downarrow \mathsf{KEqRefl}(x, G)} \qquad \frac{\Gamma, H : a = b, P : t[b] \vdash \Delta \downarrow c}{\Gamma, H : a = b, P : t[a] \vdash \Delta \downarrow \mathsf{KRewrite}(\mathit{false}, a, b, t, P, H, c)}$$

$$\frac{\Gamma, H : a = b \vdash \Delta, P : t[b] \downarrow c}{\Gamma, H : a = b \vdash \Delta, P : t[a] \downarrow \mathsf{KRewrite}(\mathit{true}, a, b, t, P, H, c)}$$

When t is a function of the form $\lambda x.\ u$, we write $t[u']$ for the substitution $u[x \mapsto u']$. These rules deal with the reflexivity of equality and the rewriting under context. They are sufficient to obtain the standard properties of equality: symmetry, transitivity, and congruence.

Application to the `rewrite` Transformation. The Why3 *rewrite* transformation is a powerful transformation that can rewrite terms modulo an equality that is under implications and universal quantifiers. It looks for a substitution to match the left-hand side of the given equality to rewrite it as the right-hand side following this substitution. Moreover, it allows rewriting from right to left instead. We instrument this transformation in the general case: using the found substitution, we define certificates to introduce in turns implications and universal quantifiers in a temporary hypothesis, to then apply symmetry of equality if needed and to use this equality to rewrite the target premise and finally remove the temporary hypothesis.

4.2 Integers

The type symbol *int*, integer literals and the operator symbols $+$, $*$, $-$, $>$, $<$, $\geq$ and $\leq$ are interpreted. To be able to certify a transformation that performs an induction on integers, we add a certificate $\mathsf{KInduction}(\mathit{ident}, \mathit{term}_{\mathit{mono}}, \mathit{term}_{\mathit{mono}}, \mathit{ident}, \mathit{ident}, \mathit{ident}, \mathit{cert}, \mathit{cert})$ to the kernel certificates with one rule for strong induction:

$$\frac{\begin{array}{c} i \text{ is fresh w.r.t. } \Gamma, \Delta, t \qquad \Sigma \Vdash i : \mathit{int} \qquad \Sigma \Vdash a : \mathit{int} \\ \Gamma, H_i : i \leq a \vdash \Delta, G : t[i] \downarrow c_{\mathit{base}} \qquad \Gamma, H_i : i > a, H_{\mathit{rec}} : \forall n : \mathit{int},\ n < i \Rightarrow t[n] \vdash \Delta, G : t[i] \downarrow c_{\mathit{rec}} \end{array}}{\Gamma \vdash \Delta, G : t[i] \downarrow \mathsf{KInduction}(i, a, t, G, H_i, H_{\mathit{rec}}, c_{\mathit{base}}, c_{\mathit{rec}})}$$

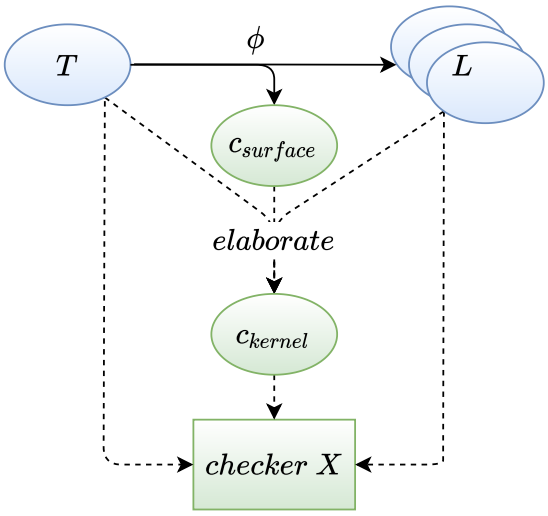
Application to the `induction` Transformation. The Why3 `induction` transformation can be called even if the context depends on the integer on which the induction is done and, in this case, the induction hypothesis takes into account this context. This transformation is instrumented to produce the certificate that first puts in the goal the premises that depend on the integer on which the induction is done, applies the KInduction certificate, and then, for the two resulting tasks, introduces those premises.

5 Certificate Checker

Let us consider an application of a certifying transformation ϕ on an initial task T that produces a certificate c and a list of resulting tasks L. To verify that this application is correct, c, T and L are provided as input to some checker. If the checker validates this application then the transformation returns L, otherwise it fails. In this section, we present the elaboration of certificates, a preprocessing step realized before calling any checker. We then show how to define such checkers and be confident in their answers.

5.1 Surface Certificates and Elaboration

Making a transformation certifying can be difficult, especially if the transformation has to produce a low level certificate. To facilitate this process, we define *surface certificates* that are easier to use than kernel certificates. We instrument the transformations to produce surface certificates instead of kernel certificates and implement an *elaboration* procedure to translate them into kernel certificates. In order to obtain the needed data to produce the kernel certificate, before calling a checker, the elaboration procedure is called with the initial task and the resulting tasks as input. Because the proof of correctness of certificates is done on the kernel certificates, we can define more complex surface certificates, as long as we are able to define the elaboration for them. Another advantage of surface certificates over kernel certificates is that they are less verbose, making them easier to produce.



Example 9. *The surface certificate* $\mathsf{SSplit}(\mathit{ident}, \mathit{cert}, \mathit{cert})$ *is elaborated into the kernel certificate* KSplit. *Suppose that a certifying transformation applied on initial task* $T := H : x_1 \vee x_2 \vdash G : x$ *returns the list* T_1; T_2 *with* $T_1 := H : x_1 \vdash G : x$ *and* $T_2 := H : x_2 \vdash G : x$ *and the surface certificate*

$$\mathsf{SSplit}(H, \mathsf{KHole}(T_1), \mathsf{KHole}(T_2))$$

The elaboration produces a kernel certificate indicating which formulas it is applied to (x_1 *and* x_2) *and that* H *is not a goal (Boolean* $false$*):*

$$\mathsf{KSplit}(false, x_1, x_2, H, \mathsf{EHole}(T_1), \mathsf{EHole}(T_2))$$

We can define every surface certificate that we find convenient. For now there are about 10 more surface certificates than kernel certificates. Among them, there are SEqSym and $\mathsf{SEqTrans}$ for symmetry and transitivity of equality and $\mathsf{SConstruct}$ described in the following example.

Example 10. *We define the surface certificate* $\mathsf{SConstruct}(\mathit{ident}, \mathit{ident}, \mathit{ident}, \mathit{cert})$ *to validate a transformation application that first merges two premises into one. More precisely, by writing* c' *a certificate* c

that has been elaborated, we should be able to derive the following rules:

$$\frac{\Gamma, P : t_1 \wedge t_2 \vdash \Delta \downarrow c'}{\Gamma, P_1 : t_1, P_2 : t_2 \vdash \Delta \downarrow \mathsf{SConstruct}(P_1, P_2, P, c)'} \qquad \frac{\Gamma \vdash \Delta, P : t_1 \vee t_2 \downarrow c'}{\Gamma \vdash \Delta, P_1 : t_1, P_2 : t_2 \downarrow \mathsf{SConstruct}(P_1, P_2, P, c)'}$$

The SConstruct *certificate does not have a corresponding kernel certificate. Instead, it is replaced during elaboration by a combination of the* KAssert *certificate on formula $t_1 \wedge t_2$ and other propositional certificates, notably* KDestruct. *Notice that we need to have access to the formula $t_1 \wedge t_2$, which is precisely the point of the elaboration and why we could not define directly* SConstruct *as a combination of surface certificates.*

5.2 OCaml Checker

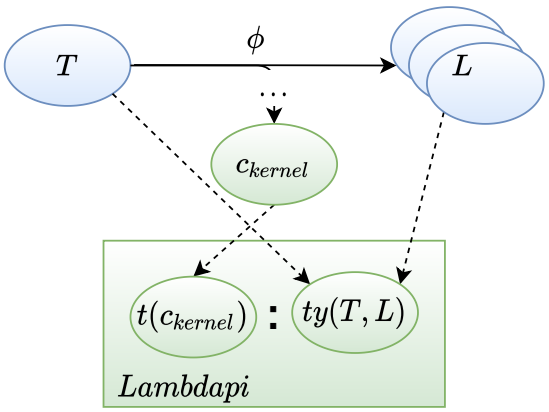
We implemented two checkers, the first one is written in OCaml and follows a computational approach: it is based on a function *ccheck* that is called with the certificate c and initial task T and interprets the certificate as instructions to derive tasks such that their validity implies the validity of T, verifying in the end that the derived tasks are the leaves of c. The checker validates the application when this function returns $true$.

Theorem 11. *Let c be the certificate produced by applying a certifying transformation on task T. If ccheck c $T = true$ then $T \downarrow c$.*

The OCaml-checker definition follows closely the semantics of the certificates. For this reason, the proof of the previous theorem is relatively straightforward to do on paper. Together with Theorem 7, this guarantees that each application of a transformation that is checked by the OCaml checker is correct.

5.3 Lambdapi Checker

The second checker uses Lambdapi/Dedukti [3], a proof assistant based on a type checker for the $\lambda\Pi$-Calculus modulo rewriting which extends the $\lambda\Pi$-Calculus formalism with custom rewriting rules. This checker has two main advantages over the OCaml checker: (1) this checker uses an off-the-shelf proof assistant, benefiting from its reliability and its features, such as the ability to define custom rewriting rules; (2) this checker is proven to be correct, and this proof is machine-checked.



Every time a transformation is called, a Lambdapi proof is generated, and this proof is then checked by the type checker. More precisely, we define a shallow embedding of proof tasks in Lambdapi: a proof task T is encoded as a Lambdapi formula $\widehat{T}$. In the diagram, a certifying transformation is applied to the initial task T and produces the resulting tasks $L := T_1; \ldots; T_n$ and a certificate c, elaborated as c_{kernel}. Our tool then generates the type $ty(T, L)$ which is the Lambdapi formula stating that $\widehat{T_1}, \widehat{T_1}, \ldots,$ and $\widehat{T_n}$ implies $\widehat{T}$, type that we call the *application correctness type*. Finally, we check that the application is correct by generating a proof term $t(c_{kernel})$ and asking Lambdapi to check that $t(c_{kernel})$ has type $ty(T, L)$.

This approach assumes that we trust Lambdapi, its type checker and the embedding of proof tasks (paragraphs 5.3.1 and 5.3.2). However the proof term generation is not contained in the trust base: the way the term is obtained does not matter as long as it has the requested type. Additionally, we have defined terms in Lambdapi for each certificate (paragraph 5.3.3), including certificates from interpreted theories (paragraph 5.3.4). These terms have been checked by Lambdapi to have the expected type so this gives us a machine-checked proof of Theorem 7.

Theorem 12. *Consider a transformation application that from task T returns the list of tasks L. If the application correctness type $ty(T,L)$ is inhabited, then this application is correct.*

5.3.1 Shallow Embedding

In Lambdapi, we define the translation of a task validity by quantifying over type symbols and function symbols, thus making these declarations explicit. We are able to quantify in this way, both at the level of types and at the level of terms, by using an encoding of the Calculus of Constructions [26] (written CoC) inside Lambdapi. We obtain a formal description of the whole task which allows us to state and prove the correctness of a transformation application. For the system to stay coherent we should be careful when adding rewriting rules and axioms (symbols in Lambdapi). We make use of an existing CoC encoding inside Lambdapi [12] to which we add the axiom of excluded middle. This encoding is also a shallow embedding inside Lambdapi, so we also get a shallow embedding of our language inside Lambdapi. In this way, we do not need to explicitly mention the context of proof and to handle it through inversion and weakening lemmas, which would make the method impracticable.

We are able to translate our whole formalism using this embedding. The translation of a term, a type or a task t inside Lambdapi is denoted $\widehat{t}$. We use exclusively the CoC syntax to describe this translation: we write $\forall x : A,\ B$ for the dependent product, $A \to B$ when B does not depend on x, $\lambda x : A,\ B$ for the abstraction and omit A when it can easily be inferred. The sorts are *Type* and *Kind*, with *Type* being of type *Kind*. To translate the terms, we use an impredicative encoding [30]. Here is an excerpt of this encoding:

$$\begin{array}{lll} \widehat{prop} & := & Type \\ \widehat{t_1 \wedge t_2} & := & \forall C : Type,\ (\widehat{t_1} \to \widehat{t_2} \to C) \to C \\ \widehat{t_1 \vee t_2} & := & \forall C : Type,\ (\widehat{t_1} \to C) \to (\widehat{t_2} \to C) \to C \end{array} \qquad \begin{array}{lll} \widehat{\bot} & := & \forall C : Type,\ C \\ \widehat{\top} & := & \widehat{\bot} \to \widehat{\bot} \\ \widehat{\neg t} & := & \widehat{t} \to \widehat{\bot} \end{array}$$

We note $\widehat{\neg}\ u$ for $u \to \widehat{\bot}$ such that $\widehat{\neg t} = \widehat{\neg}\ \widehat{t}$ and we extend this notation to the conjunction and the disjunction. Note that $\widehat{\top}$ is inhabited by $\lambda c,\ c$.

5.3.2 Translating Tasks

Let us give the translation of a task, where $Type^n$ denotes the n-ary function over *Type* (for example, $Type^2$ is $Type \to Type \to Type$).

Task $I \mid \Sigma \mid \Gamma \vdash \Delta$ with :	Corresponding Lambdapi term:
$I := \iota_1 : i_1, \ldots, \iota_m : i_m$	$\forall \iota_1 : Type^{i_1},\ \ldots, \forall \iota_m : Type^{i_m},$
$\Sigma := f_1 : \tau_1, \ldots, f_n : \tau_n$	$\forall f_1 : \widehat{\tau_1},\ \ldots, \forall f_n : \widehat{\tau_n},$
$\Gamma := H_1 : t_1, \ldots, H_k : t_k$	$\widehat{t_1} \to \cdots \to \widehat{t_k} \to$
$\Delta := G_1 : u_1, \ldots, G_l : u_l$	$\widehat{\neg u_1} \to \cdots \to \widehat{\neg u_l} \to \widehat{\bot}$

Note that for polymorphic symbols, we need to declare them with extra type parameters and to apply them to the appropriate type in the translation.

5.3.3 Proof Term

For each kernel rule, we associate a Lambdapi type and define a term that has this type. When building a proof term, we first introduce the identifiers of the resulting tasks and the identifiers of the type symbols, function symbols and name of the premises of the initial task. Then, we translate the whole certificate using the terms having the associated types. The KHole certificate is special and does not have a fixed associated type. Instead, it is translated as the identifier of the task it contains applied to its symbols and premises following the same order that they have been introduced in. Assuming that our encoding is correct, the fact that we define a term for every rule of the certificate semantics gives us a machine-checked proof of the certificate correctness, Theorem 7.

To produce the proof term, we benefit from the elaboration of certificate in two ways. First, the fact that the kernel certificates are elementary also facilitates the definition of the terms corresponding to a kernel rule. Second, each kernel certificate comes with additional data that we are able to use to define such terms.

Example 13. *For the* KSplit *rule presented in Figure 2, we define a Lambdapi term split that has the associated type* $\forall t_1 : Type,\ \forall t_2 : Type,\ (t_1 \to \widehat{\bot}) \to (t_2 \to \widehat{\bot}) \to t_1 \mathbin{\widehat{\vee}} t_2 \to \widehat{\bot}$. *We check the application of Example 9 by verifying that the type*

$$\begin{aligned}&(\forall x_1,\ \forall x,\ x_1 \to \widehat{\neg}\, x \to \widehat{\bot}) \to\\ &(\forall x_2,\ \forall x,\ x_2 \to \widehat{\neg}\, x \to \widehat{\bot}) \to\\ &\ \forall x_1,\ \forall x_2,\ \forall x,\ x_1 \mathbin{\widehat{\vee}} x_2 \to \widehat{\neg}\, x \to \widehat{\bot}\end{aligned}$$

is inhabited by the term

$$\begin{aligned}&\lambda s_1,\ \lambda s_2,\ \lambda x_1,\ \lambda x_2,\ \lambda x,\ \lambda H,\ \lambda G,\\ &split\ x_1\ x_2\ (\lambda H,\ s_1\ x_1\ x\ H\ G)\ (\lambda H,\ s_2\ x_2\ x\ H\ G)\end{aligned}$$

Notice that split takes the formulas it is applied to as arguments (x_1 and x_2) and that those formulas have been found by elaborating the certificate.

5.3.4 Encoding of Interpreted Theories

In Lambdapi, interpreted symbols are first declared in the preamble. When interpreted symbols have corresponding certificate rules, we need to use the properties of those symbols to prove that the types associated to these rules are inhabited. Instead of declaring such symbols, we define them, which allows us to prove the needed properties. Since our Lambdapi development is included in the trusted code base of the Lambdapi checker, we make sure to only add axioms and rewrite rules when necessary.

Polymorphic Equality. We define the equality in Lambdapi using the Leibniz definition of equality: two terms t_1 and t_2 of type τ are equal when $\forall Q : \tau \to Type,\ Q\ t_1 \to Q\ t_2$. Note that the context of rewriting in the KRewrite rules is explicitly given as a function. We use this function, translated as a Lambdapi function, to apply it to the Leibniz equality when writing a proof term for a KRewrite certificate.

number of variables	5	10	15	20	25	50	100	200	400	800
transformation time (sec)	~ 0	~ 0	0.008	0.016	0.020	0.080	0.29	1.21	5.5	25
kernel certificate size (kB)	2.1	5.8	12	19	28	85	270	950	3500	13000
OCaml checker time (sec)	~ 0	~ 0	~ 0	~ 0	~ 0	~ 0	~ 0	0.020	0.084	0.35
Lambdapi checker time (sec)	0.072	0.25	0.80	2.0	4.2	48	660	-	-	-

Figure 3: Tests on Propositional Tasks

Integers. We define the integer type and usual integer operators in Lambdapi. First, we define binary positive integers and binary negative integers. Integers are then either 0, positive or negative. We use rewrite rules to define those data types which should be understood as algebraic data types [5]. From these definitions, we get a simple induction principle that we use to define a Lambdapi term corresponding to the stronger induction principle described by the rule of the certificate KInduction.

6 Experimental Evaluation

The goal of this article is to provide a practical framework to render logical transformations certifying. We apply the framework to Why3 and show that the approach does not have an inherent problem of efficiency. More importantly, we show that it is expressive enough to allow us to render complex transformations certifying.

6.1 Tests and Benchmarks

We defined simple certifying transformations (about 15 of them) to test every certificate. We also defined a more complex transformation called `blast` meant to discharge tautological propositional tasks. This transformation decomposes every logical connector appearing at the head of formulas before calling itself recursively. Rendering this transformation certifying required using the composition of certifying transformations. We evaluated the efficiency of our checkers by applying this transformation on problems of increasing size in Figure 3. The problem with n propositional variables is to verify that:

$$p_1 \Rightarrow (p_1 \Rightarrow p_2) \Rightarrow \ldots (p_{n-1} \Rightarrow p_n) \Rightarrow p_n$$

We notice that the size of the kernel certificates is not linear with respect to the number of variables. This is due to the fact that, contrary to the surface certificates, the kernel certificates contain the formulas they are applied to. Looking at the OCaml checker, our approach does not seem to have an inherent problem of efficiency as the overhead it adds to the transformation is negligible. On the other hand, the Lambdapi checker seems to be much slower. Performances of Lambdapi have already been improved for our purposes [16–20], and we believe they could be further improved in future versions. We could also modify our checker to help Lambdapi, for example by abstracting away big formulas. We leave this for future work.

6.2 Applications

We evaluated our method by applying it at different levels. When rendering the existing transformations `rewrite` and `induction` certifying, we found that it is well-suited to add interpreted theories. When transformations do not specifically deal with the theory in question, we do not need to extend our certificate format, while, in general, the duo surface/kernel certificates allows us to only define a minimal set of kernel rules, even if it means deriving more surface certificates. By defining the Lambdapi checker, we gave a machine-checked proof of the rules of our certificates which gives us confidence in our certificates and their semantics.

6.3 Better Understanding of Transformations

This work has led to a better understanding of transformations. On one hand, instrumenting a transformation to produce an appropriate certificate requires to understand why each application of this transformation is correct. Additionally, once this is done, reading the certificate gives us another way to understand why a particular transformation application is correct. On the other hand, this work had led to the definition of the semantics of tasks inside Lambdapi and the definition of the correction of a transformation in this setting.

In particular, type quantification is explicit in Section 2.1. For example, the formula $\Pi\alpha.\ (\forall x : \alpha.\ \forall y : \alpha.\ x = y) \vee \neg(\forall x : \alpha.\ \forall y : \alpha.\ x = y)$ means that every type α either has at most one element or it has more than one. This formula is provable but we cannot apply the certificate KSplit on such an hypothesis. By contrast, in Why3, the type quantification is implicit, and it is possible apply the `destruct` transformation on the hypothesis. This gives us two resulting tasks: one with an hypothesis which states that every type has at most one element, and the other with an hypothesis which states that every type has more than one element, both being contradictory. This bug [15] has been found in the transformation `destruct` when encoding proof tasks in Lambdapi; a similar bug was also found in the transformation `case`.

7 Related Work

To aid deductive program verification, a number of tools have been developed, based on proof assistants or independently from them. In the first case, the programming language on which the verification is done is built from dedicated libraries and definition of both the programming language and its logic inside the proof assistant. This is the case for example for the library Iris [27] built on top of Coq and that allows reasoning about concurrent, imperative programs or the library AutoCorres [23,24] built on top of Isabelle and allowing to verify C programs. In such context, the correctness of the approach is based on the formal semantics of programs and on deduction rules established once and for all, which requires a large proof effort, thus limiting the flexibility of the language. In the second case, the tools developed are verifying annotated programs, and generate proof obligations that are discharged by automatic theorem provers such as SMT solvers. Examples of such tools are Why3, Dafny, Viper, Frama-C and SPARK. Even though they rely on strong fundamental bases, their particular implementations of such tools and some practical aspects such as their use of automatic theorem provers have not been machine-checked and can contain bugs. An exception is given by F* [31], whose encoding's correctness to SMT logic has been partially proved in Coq [1].

Our work lies in between these two approaches. On one hand logical transformations are similar to tactics used in proof assistants such as Coq [13], except that our transformations are considered part of the

trusted code base. On the other hand, logical transformations can be used automatically or interactively to help discharging proof obligations. We followed a skeptical approach extended with a preprocessing step (namely the elaboration of certificates) similarly to [10], except that our framework allows to check higher order proofs, and that the focus is put on the ease of production of certificates. Indeed, we aim at making it as easy as possible to render transformations certifying. We designed two checkers: one based on the reflexive approach, known to be very efficient [2, 25] and the other one based on a shallow embedding into the Lambdapi proof assistant. When using a shallow embedding, the correctness of the verification relies on the considered proof tool's correctness which makes its proof much easier [9, 11].

8 Conclusion

We presented a framework to validate logical transformations based on a skeptical approach. When defining certificates, we put an emphasis on *modularity* by having certificates with holes and, with the notions of surface and kernel certificates, *ease of use* without compromising the checker's verification. We combined all of these notions and applied them to Why3 by implementing the certificate generation for various transformations and the certificate verification with two checkers. The first checker was written in OCaml and uses a computational approach which makes it very efficient while the second checker is based on Lambdapi and gives us formal guarantees to its correctness. We extended our work by adding the interpreted theories of the integers and of the polymorphic equality. This allowed us to instrument more complex and existing transformations to produce certificates, such as `induction` and `rewrite`. Finally, we validated our method during development and through tests and benchmarks.

Future Work. The current application of our method to Why3 could be improved at different levels. The first idea is to instrument more transformations to produce certificates, with polymorphism elimination [7] and algebraic data type elimination being important challenges. As the number of certifying transformations increases, we also want to improve the efficiency of the verification. To do so, we consider two factors: first, we want to compress certificates on the fly when combining them; second, we want to improve the efficiency of the Lambdapi checker by allowing to reuse the context of proof that does not change. Additionally, we consider adding support for more interpreted theories, while keeping the number of axioms and rewrite rules added to Lambdapi to a minimum.

A long term goal is to increase trust in other parts of Why3. For example, we could improve trust when calling automatic theorem provers [2, 8] or improve trust in the proof task generation which would require to formalize the semantics of the Why3 programming language [14].

Finally, our method is not specific to Why3 and can be applied, in general, to certified logical encodings. In particular, existing (certifying) transformations could be used for encoding a proof assistant's logic into an automatic theorem prover's logic in order to benefit from both systems.

Acknowledgments. We are grateful to Alexandrina Korneva for the English proofreading and to Claude Marché, Chantal Keller and Andrei Paskevich for their constant support and their helpful suggestions.

References

[1] Alejandro Aguirre (2016): *Towards a provably correct encoding from F* to SMT*. Master's thesis. Available at `https://prosecco.gforge.inria.fr/personal/hritcu/students/alejandro/report.pdf`.

[2] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Thery & Benjamin Werner (2011): *A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses*. In: *First International Conference on Certified Programs and Proofs*, doi:10.1007/978-3-642-25379-9_12.

[3] Ali Assaf, Guillaume Burel, Raphal Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant & Ronan Saillard (2016): *Expressing theories in the $\lambda\Pi$-calculus modulo theory and in the Dedukti system*. In: *22nd International Conference on Types for Proofs and Programs*. Available at `https://hal-mines-paristech.archives-ouvertes.fr/hal-01441751`.

[4] Henk Barendregt & Erik Barendsen (2002): *Autarkic computations in formal proofs*. *Journal of Automated Reasoning*, doi:10.1023/A:1015761529444.

[5] Frédéric Blanqui (2003): *Inductive Types in the Calculus of Algebraic Constructions*. Lecture Notes in Computer Science, doi:10.1007/3-540-44904-3_4.

[6] François Bobot, Jean-Christophe Filliâtre, Claude Marché & Andrei Paskevich (2015): *Let's Verify This with Why3*. *International Journal on Software Tools for Technology Transfer (STTT)*, doi:10.1007/s10009-014-0314-5.

[7] François Bobot & Andrei Paskevich (2011): *Expressing Polymorphic Types in a Many-Sorted Language*, doi:10.1007/978-3-642-24364-6_7.

[8] Sascha Böhme & Tjark Weber (2010): *Fast LCF-Style Proof Reconstruction for Z3*. In: *Interactive Theorem Proving*, doi:10.1007/978-3-642-14052-5_14.

[9] Raphaël Cauderlier & Pierre Halmagrand (2015): *Checking Zenon Modulo Proofs in Dedukti*. In: *Proof eXchange for Theorem Proving*, doi:10.4204/EPTCS.186.7.

[10] Zakaria Chihani, Dale Miller & Fabien Renaud (2013): *Checking Foundational Proof Certificates for First-Order Logic (Extended Abstract)*. In: *PxTP 2013. Third International Workshop on Proof Exchange for Theorem Proving*, doi:10.29007/7gnr.

[11] Évelyne Contejean (2008): *Coccinelle, a Coq library for rewriting*. In: *Types*.

[12] Denis Cousineau & Gilles Dowek (2007): *Embedding Pure Type Systems in the lambda-Pi-calculus modulo*. In: *Typed lambda calculi and applications*, doi:10.1007/978-3-540-73228-0_9.

[13] David Delahaye (2000): *A tactic language for the system Coq*. In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, doi:10.1007/3-540-44404-1_7.

[14] Jean-Christophe Filliâtre (1999): *Preuve de programmes impératifs en théorie des types*. Ph.D. thesis. Available at `http://www.lri.fr/~filliatr/ftp/publis/these.ps.gz`.

[15] Quentin Garchery (2021): *destruct/case transformations incorrectly handle polymorphic formulas*. `https://gitlab.inria.fr/why3/why3/-/issues/525/`.

[16] Quentin Garchery (2021): *Performance slowdown with variable shadowing*. `https://github.com/Deducteam/lambdapi/issues/565`.

[17] Quentin Garchery (2021): *Performance with growing context*. `https://github.com/Deducteam/lambdapi/issues/579`.

[18] Quentin Garchery (2021): *Performances with both new variables and hypotheses in context*. `https://github.com/Deducteam/lambdapi/issues/595`.

[19] Quentin Garchery (2021): *Performances with linear propositional problem*. `https://github.com/Deducteam/lambdapi/issues/649`.

[20] Quentin Garchery (2021): *Performances with nested applications in context*. `https://github.com/Deducteam/lambdapi/issues/584`.

[21] Quentin Garchery (2021): *Why3 cert_pxtp branch*. Available at `https://gitlab.inria.fr/why3/why3/-/blob/cert_pxtp/README_PXTP.md`.

[22] Quentin Garchery, Chantal Keller, Claude Marché & Andrei Paskevich (2020): *Des transformations logiques passent leur certicat*. In: *JFLA 2020 - Journées Francophones des Langages Applicatifs*, Gruissan, France. Available at `https://hal.inria.fr/hal-02384946`.

[23] David Greenaway (2015): *Automated proof-producing abstraction of C code*. Ph.D. thesis, CSE, UNSW. Available at `http://unsworks.unsw.edu.au/fapi/datastream/unsworks:13743/SOURCE02?view=true`.

[24] David Greenaway, Japheth Lim, June Andronick & Gerwin Klein (2014): *Don't Sweat the Small Stuff: Formal Verification of C Code Without the Pain*. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*, doi:10.1145/2594291.2594296.

[25] Benjamin Grégoire, Laurent Théry & Benjamin Werner (2006): *A Computational Approach to Pocklington Certificates in Type Theory*. In: *Functional and Logic Programming*, doi:10.1007/11737414_8.

[26] Gérard P. Huet (1987): *The Calculus of Constructions: State of the Art*. In: *Foundations of Software Technology and Theoretical Computer Science*, doi:10.1007/3-540-18625-5_61.

[27] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer & Lars Birkedal (2017): *The Essence of Higher-Order Concurrent Separation Logic*. In: *26th European Symposium on Programming Languages and Systems*, doi:10.1007/978-3-662-54434-1_26.

[28] Stéphane Lescuyer (2011): *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq. (Formalisation et developpement d'une tactique reflexive pour la demonstration automatique en coq)*. Ph.D. thesis, University of Paris-Sud, Orsay, France. Available at `https://tel.archives-ouvertes.fr/tel-00713668`.

[29] Robin Milner (1978): *A theory of type polymorphism in programming*. *Journal of computer and system sciences*, doi:10.1016/0022-0000(78)90014-4.

[30] Frank Pfenning & Christine Paulin-Mohring (1989): *Inductively Defined Types in the Calculus of Constructions*. Lecture Notes in Computer Science, doi:10.1007/BFb0040259.

[31] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue & Santiago Zanella Béguelin (2016): *Dependent types and multi-monadic effects in F**. In: *Principles of Programming Languages*, doi:10.1145/2837614.2837655.

A Typing

The predicate $\Sigma \Vdash t : \tau$ holds when t has no free type variables and is of type τ in signature Σ and is formally defined by the following rules:

$$\frac{I, \iota : 0 \mid \Sigma \Vdash t[\alpha \mapsto \iota] : \mathit{prop} \qquad \iota \notin I}{I \mid \Sigma \Vdash (\Pi \alpha.\, t) : \mathit{prop}} \qquad \frac{\tau \text{ is a subtype of } \Sigma(x) \qquad \tau \text{ has no type variables}}{\Sigma \Vdash x : \tau}$$

$$\frac{}{\Sigma \Vdash \top : \mathit{prop}} \qquad \frac{}{\Sigma \Vdash \bot : \mathit{prop}} \qquad \frac{\Sigma \Vdash t : \mathit{prop}}{\Sigma \Vdash \neg t : \mathit{prop}} \qquad \frac{\Sigma \Vdash t_1 : \mathit{prop} \qquad \Sigma \Vdash t_2 : \mathit{prop}}{\Sigma \Vdash t_1 \; \mathit{op} \; t_2 : \mathit{prop}}$$

$$\frac{\Sigma \Vdash t_1 : \tau' \rightsquigarrow \tau \qquad \Sigma \Vdash t_2 : \tau'}{\Sigma \Vdash t_1 \; t_2 : \tau} \qquad \frac{\Sigma, x : \tau \Vdash t : \mathit{prop} \qquad \tau \text{ has no type variables} \qquad x \notin \Sigma}{\Sigma \Vdash (\forall x : \tau.\, t) : \mathit{prop}}$$

$$\frac{\Sigma, x : \tau \Vdash t : \mathit{prop} \qquad \tau \text{ has no type variables} \qquad x \notin \Sigma}{\Sigma \Vdash (\exists x : \tau.\, t) : \mathit{prop}}$$

$$\frac{\Sigma, x : \tau' \Vdash t : \tau \qquad \tau' \text{ has no type variables} \qquad x \notin \Sigma}{\Sigma \Vdash (\lambda x : \tau'.\, t) : \tau' \rightsquigarrow \tau}$$

B Certificate rules

For each kernel certificate appearing in this article, we give its corresponding rules. Theses rules are taken from the set of rules defininig the predicate $T \downarrow c$.

$$\frac{}{\Gamma \vdash \Delta \downarrow \mathsf{KHole}(\Gamma \vdash \Delta)} \qquad \frac{}{\Gamma, H : \bot \vdash \Delta \downarrow \mathsf{KTrivial}(\mathit{false}, H)} \qquad \frac{}{\Gamma \vdash \Delta, G : \top \downarrow \mathsf{KTrivial}(\mathit{true}, G)}$$

$$\frac{\Sigma \mid \Gamma \vdash \Delta, P : t \downarrow c_1 \qquad \Sigma \mid \Gamma, P : t \vdash \Delta \downarrow c_2 \qquad \Sigma \Vdash t : \mathit{prop}}{\Sigma \mid \Gamma \vdash \Delta \downarrow \mathsf{KAssert}(P, t, c_1, c_2)}$$

$$\frac{\Gamma, H : t_1 \vdash \Delta \downarrow c_1 \qquad \Gamma, H : t_2 \vdash \Delta \downarrow c_2}{\Gamma, H : t_1 \vee t_2 \vdash \Delta \downarrow \mathsf{KSplit}(\mathit{false}, t_1, t_2, H, c_1, c_2)} \qquad \frac{\Gamma \vdash \Delta, G : t_1 \downarrow c_1 \qquad \Gamma \vdash \Delta, G : t_2 \downarrow c_2}{\Gamma \vdash \Delta, G : t_1 \wedge t_2 \downarrow \mathsf{KSplit}(\mathit{true}, t_1, t_2, G, c_1, c_2)}$$

$$\frac{\Gamma, H_1 : t_1, H_2 : t_2 \vdash \Delta \downarrow c}{\Gamma, H : t_1 \wedge t_2 \vdash \Delta \downarrow \mathsf{KDestruct}(\mathit{false}, t_1, t_2, H, H_1, H_2, c)}$$

$$\frac{\Gamma \vdash \Delta, G_1 : t_1, G_2 : t_2 \downarrow c}{\Gamma \vdash \Delta, G : t_1 \vee t_2 \downarrow \mathsf{KDestruct}(\mathit{true}, t_1, t_2, G, G_1, G_2, c)}$$

$$\frac{\Sigma, y : \tau \mid \Gamma, H : t[x \mapsto y] \vdash \Delta \downarrow c \qquad y \text{ is fresh w.r.t. } \Sigma, \Gamma, \Delta, t}{\Sigma \mid \Gamma, H : \exists x : \tau.\ t \vdash \Delta \downarrow \mathsf{KIntroQuant}(\mathit{false}, \tau, \lambda x : \tau.\ t, H, y, c)}$$

$$\frac{\Sigma, y : \tau \mid \Gamma \vdash \Delta, G : t[x \mapsto y] \downarrow c \qquad y \text{ is fresh w.r.t. } \Sigma, \Gamma, \Delta, t}{\Sigma \mid \Gamma \vdash \Delta, G : \forall x : \tau.\ t \downarrow \mathsf{KIntroQuant}(\mathit{true}, \tau, \lambda x : \tau.\ t, G, y, c)}$$

$$\frac{\Sigma \mid \Gamma, H_1 : \forall x : \tau.\ t, H_2 : t[x \mapsto u] \vdash \Delta \downarrow c \qquad \Sigma \Vdash u : \tau}{\Sigma \mid \Gamma, H_1 : \forall x : \tau.\ t \vdash \Delta \downarrow \mathsf{KInstQuant}(\mathit{false}, \tau, \lambda x : \tau.\ t, H_1, H_2, u, c)}$$

$$\frac{\Sigma \mid \Gamma \vdash \Delta, G_1 : \exists x : \tau.\ t, G_2 : t[x \mapsto u] \downarrow c \qquad \Sigma \Vdash u : \tau}{\Sigma \mid \Gamma \vdash \Delta, G_1 : \exists x : \tau.\ t \downarrow \mathsf{KInstQuant}(\mathit{true}, \tau, \lambda x : \tau.\ t, G_1, G_2, u, c)}$$

$$\frac{I, \iota : 0 \mid \Sigma \mid \Gamma \vdash \Delta, G : t[\alpha \mapsto \iota] \downarrow c \qquad \iota \notin I}{I \mid \Sigma \mid \Gamma \vdash \Delta, G : \Pi\alpha.\ t \downarrow \mathsf{KIntroType}(\Pi\alpha.\ t, G, \iota, c)}$$

$$\frac{\Gamma, H_1 : \Pi\alpha.\ t, H_2 : t[\alpha \mapsto \tau] \vdash \Delta \downarrow c \qquad \tau \text{ has no type variables}}{\Gamma, H_1 : \Pi\alpha.\ t \vdash \Delta \downarrow \mathsf{KInstType}(\Pi\alpha.\ t, H_1, H_2, \tau, c)}$$

$$\frac{}{\Sigma \mid \Gamma \vdash \Delta, G : x = x \downarrow \mathsf{KEqRefl}(x, G)} \qquad \frac{\Gamma, H : a = b, P : t[b] \vdash \Delta \downarrow c}{\Gamma, H : a = b, P : t[a] \vdash \Delta \downarrow \mathsf{KRewrite}(\mathit{false}, a, b, t, P, H, c)}$$

$$\frac{\Gamma, H : a = b \vdash \Delta, P : t[b] \downarrow c}{\Gamma, H : a = b \vdash \Delta, P : t[a] \downarrow \mathsf{KRewrite}(\mathit{true}, a, b, t, P, H, c)}$$

$$\frac{\begin{array}{c} i \text{ is fresh w.r.t. } \Gamma, \Delta, t \qquad \Sigma \Vdash i : \mathit{int} \qquad \Sigma \Vdash a : \mathit{int} \\ \Gamma, H_i : i \leq a \vdash \Delta, G : t[i] \downarrow c_{base} \qquad \Gamma, H_i : i > a, H_{rec} : \forall n : \mathit{int},\ n < i \Rightarrow t[n] \vdash \Delta, G : t[i] \downarrow c_{rec} \end{array}}{\Gamma \vdash \Delta, G : t[i] \downarrow \mathsf{KInduction}(i, a, t, G, H_i, H_{rec}, c_{base}, c_{rec})}$$