Static and User-Extensible Proof Checking

Antonis Stampoulis   Zhong Shao

Department of Computer Science
Yale University
New Haven, CT 06520, USA
{antonis.stampoulis,zhong.shao}@yale.edu

Abstract

Despite recent successes, large-scale proof development within proof assistants remains an arcane art that is extremely time-consuming. We argue that this can be attributed to two profound shortcomings in the architecture of modern proof assistants. The first is that proofs need to include a large amount of minute detail; this is due to the rigidity of the proof checking process, which cannot be extended with domain-specific knowledge. In order to avoid these details, we rely on developing and using tactics, specialized procedures that produce proofs. Unfortunately, tactics are both hard to write and hard to use, revealing the second shortcoming of modern proof assistants. This is because there is no static knowledge about their expected use and behavior.

As has recently been demonstrated, languages that allow type-safe manipulation of proofs, like Beluga, Delphin and VeriML, can be used to partly mitigate this second issue, by assigning rich types to tactics. Still, the architectural issues remain. In this paper, we build on this existing work, and demonstrate two novel ideas: an *extensible conversion rule* and support for *static proof scripts*. Together, these ideas enable us to support both user-extensible proof checking, and sophisticated static checking of tactics, leading to a new point in the design space of future proof assistants. Both ideas are based on the interplay between a light-weight staging construct and the rich type information available.

Categories and Subject Descriptors   D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms   Languages, Verification

1. Introduction

There have been various recent successes in using proof assistants to construct foundational proofs of large software, like a C compiler [Leroy 2009] and an OS microkernel [Klein et al. 2009], as well as complicated mathematical proofs [Gonthier 2008]. Despite this success, the process of large-scale proof development using the foundational approach remains a complicated endeavor that requires significant manual effort and is plagued by various architectural issues.

The big benefit of using a foundational proof assistant is that the proofs involved can be checked for validity using a very small proof checking procedure. The downside is that these proofs are very large, since proof checking is fixed. There is no way to add domain-specific knowledge to the proof checker, which would enable proofs that spell out less details. There is good reason for this, too: if we allowed arbitrary extensions of the proof checker, we could very easily permit it to accept invalid proofs.

Because of this lack of extensibility in the proof checker, users rely on tactics: procedures that produce proofs. Users are free to write their own tactics, that can create domain-specific proofs. In fact, developing domain-specific tactics is considered to be good engineering when doing large developments, leading to significantly decreased overall effort – as shown, e.g. in Chlipala [2011]. Still, using and developing tactics is error-prone. Tactics are essentially untyped functions that manipulate logical terms, and thus tactic programming is untyped. This means that common errors, like passing the wrong argument, or expecting the wrong result, are not caught statically. Exacerbating this, proofs contained within tactics are not checked statically, when the tactic is defined. Therefore, even if the tactic is used correctly, it could contain serious bugs that manifest only under some conditions.

With the recent advent of programming languages that support strongly typed manipulation of logical terms, such as Beluga [Pientka and Dunfield 2008], Delphin [Poswolsky and Schürmann 2008] and VeriML [Stampoulis and Shao 2010], this situation can be somewhat mitigated. It has been shown in Stampoulis and Shao [2010] that we can specify what kinds of arguments a tactic expects and what kind of proof it produces, leading to a type-safe programming style. Still, this does not address the fundamental problem of proof checking being fixed – users still have to rely on using tactics. Furthermore, the proofs contained within the type-safe tactics are in fact proof-producing programs, which need to be evaluated upon invocation of the tactic. Therefore proofs within tactics are not checked statically, and they can still cause the tactics to fail upon invocation.

In this paper, we build on the past work on these languages, aiming to solve both of these issues regarding the architecture of modern proof assistants. We introduce two novel ideas: support for an **extensible conversion rule** and **static proof scripts** inside tactics. The former technique enables proof checking to become user-extensible, while maintaining the guarantee that only logically sound proofs are admitted. The latter technique allows for statically checking the proofs contained within tactics, leading to increased guarantees about their runtime behavior. Both techniques are based on the same mechanism, which consists of a light-weight staging construct. There is also a deep synergy between them, allowing us to use the one to the benefit of the other.

Our main contributions are the following:

- First, we present what we believe is the first technique for having an extensible conversion rule, which combines the following characteristics: it is safe, meaning that it preserves logical soundness; it is user-extensible, using a familiar, generic pro-

POPL'12,   January 25–27, 2012, Philadelphia, PA, USA.

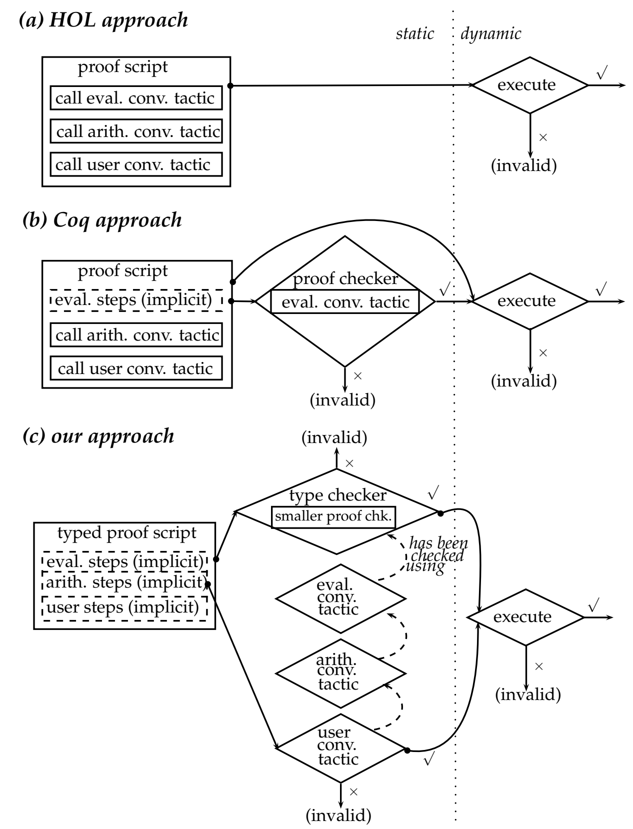

Figure 1. Checking proof scripts in various proof assistants

gramming model; and, it does not require metatheoretic additions to the logic, but can be used to simplify the logic instead.

- Second, building on existing work for typed tactic development, we introduce static checking of the proof scripts contained within tactics. This significantly reduces the development effort required, allowing us to write tactics that benefit from existing tactics and from the rich type information available.
- Third, we show how typed proof scripts can be seen as an alternative form of proof witness, which falls between a proof object and a proof script. Receivers of the certificate are able to decide on the tradeoff between the level of trust they show and the amount of resources needed to check its validity.

In terms of technical contributions, we present a number of technical advances in the metatheory of the aforementioned programming languages. These include a simple staging construct that is crucial to our development and a new technique for variable representation. We also show a condition under which static checking of proof scripts inside tactics is possible. Last, we have extended an existing prototype implementation with a significant number of features, enabling it to support our claims, while also rendering its use as a proof assistant more practical.

2. Informal presentation

Glossary of terms. We will start off by introducing some concepts that will be used throughout the paper. The first fundamental concept we will consider is the notion of a *proof object*: given a derivation of a proposition inside a formal logic, a proof object is a term representation of this derivation. A *proof checker* is a program that can decide whether a given proof object is a valid derivation of a specific proposition or not. Proof objects are extremely verbose and are thus hard to write by hand. For this reason, we use *tactics*: functions that produce proof objects. By combining tactics together, we create proof-producing programs, which we call *proof scripts*. If a proof script is evaluated, and the evaluation completes successfully, the resulting proof object can be checked using the original proof checker. In this way, the trusted base of the system is kept at the absolute minimum. The language environment where proof scripts and tactics are written and evaluated is called a *proof assistant*; evidently, it needs to include a proof checker.

Checking proof objects. In order to keep the size of proof objects manageable, many of the logics used for mechanized proof checking include a *conversion rule*. This rule is used implicitly by the proof checker to decide whether any two propositions are equivalent; if it determines that they are indeed so, the proof of their equivalence can be omitted. We can thus think of it as a special tactic that is embedded within the proof checker, and used implicitly.

The more sophisticated the relation supported by the conversion rule is, the simpler are proof objects to write, since more details can be omitted. On the other hand, the proof checker becomes more complicated, as does the metatheory proof showing the soundness of the associated logic. The choice in Coq [Barras et al. 2010], one of the most widely used proof assistants, with respect to this trade-off, is to have a conversion rule that identifies propositions up to evaluation. Nevertheless, extended notions of conversion are desirable, leading to proposals like CoqMT [Strub 2010], where equivalence up to first-order theories is supported. In both cases, the conversion rule is fixed, and extending it requires significant amounts of work. It is thus not possible for users to extend it using their own, domain-specific tactics, and proof objects are thus bound to get large. This is why we have to resort to writing proof scripts.

Checking proof scripts. As mentioned earlier, in order to validate a proof script we need to evaluate it (see Fig. 1a); this is the modus operandi in proof assistants of the HOL family [Harrison 1996; Slind and Norrish 2008]. Therefore, it is easy to extend the checking procedure for proof scripts by writing a new tactic, and calling it as part of a script. The price that this comes to is that there is no way to have any sort of static guarantee about the validity of the script, as proof scripts are completely untyped. This can be somewhat mitigated in Coq by utilizing the static checking that it already supports: the proof checker, and especially, the conversion rule it contains (see Fig. 1b). We can employ proof objects in our scripts; this is especially useful when the proof objects are trivial to write but trigger complex conversion checks. This is the essential idea behind techniques like proof-by-reflection [Boutin 1997], which lead to more robust proof scripts.

In previous work [Stampoulis and Shao 2010] we introduced VeriML, a language that enables programming tactics and proof scripts in a typeful manner using a general-purpose, side-effectful programming model. Combining typed tactics leads to *typed proof scripts*. These are still programs producing proof objects, but the proposition they prove is carried within their type. Information about the current proof state (the set of hypotheses and goals) is also available statically at every intermediate point of the proof script. In this way, the static assurances about proof scripts are significantly increased and many potential sources of type errors are removed. On the other hand, the proof objects contained within the scripts are still checked using a fixed proof checker; this ultimately means that the set of possible static guarantees is still fixed.

Extensible conversion rule. In this paper, we build on our earlier work on VeriML. In order to further increase the amount of static checking of proof scripts that is possible within this language, we propose the notion of an extensible conversion rule (see Fig. 1c). It enables users to write their own domain-specific conversion checks

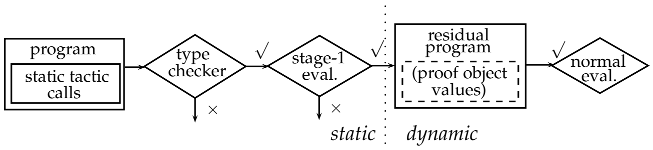

Figure 2. Staging in VeriML

$$t ::= \textit{proof object constructors} \mid \textit{propositions}$$
$$\mid \textit{natural numbers, lists, etc.} \mid \textit{sorts and types} \mid X/\sigma$$
$$\Phi ::= \bullet \mid \Phi,\, x : t \qquad T ::= [\Phi]\, t$$
$$\Psi ::= \bullet \mid \Psi,\, X : T \qquad \sigma ::= \bullet \mid \sigma,\, x \mapsto t$$
$$\text{main judgement: } \Psi;\, \Phi \vdash t : t' \quad \textit{(type of a logical term)}$$

Figure 3. Assumptions about the logic language

that get included in the conversion rule. This leads to simpler proof scripts, as more parts of the proof can be inferred by the conversion rule and can therefore be omitted. Also, it leads to increased static guarantees for proof scripts, since the conversion checks happen before the rest of the proof script is evaluated.

The way we achieve this is by programming the conversion checks as type-safe tactics within VeriML, and then evaluating them statically using a simple staging mechanism (see Fig. 2). The type of the conversion tactics requires that they produce a proof object which proves the claimed equivalence of the propositions. In this way, type safety of VeriML guarantees that soundness is maintained. At the same time, users are free to extend the conversion rule with their own conversion tactics written in a familiar programming model, without requiring any metatheoretic additions or termination proofs. Such proofs are only necessary if decidability of the extra conversion checks is desired. Furthermore, this approach allows for metatheoretic reductions as the original conversion rule can be programmed within the language. Thus it can be removed from the logic, and replaced by the simpler notion of explicit equalities, leading to both simpler metatheory and a smaller trusted base.

Checking tactics. The above approach addresses the issue of being able to extend the amount of static checking possible for proof scripts. But what about tactics? Our existing work on VeriML shows how the increased type information addresses some of the issues of tactic development using current proof assistants, where tactics are programmed in a completely untyped manner.

Still, if we consider the case of tactics more closely, we will see that there is a limitation to the amount of checking that is done statically, even using this language. When programming a new tactic, we would like to reuse existing tactics to produce the required proofs. Therefore, rather than writing proof objects by hand inside the code of a tactic, we would rather use proof scripts. The issue is that in order to check whether the contained proof scripts are valid, they need to be evaluated – but this only happens when an invocation of the tactic reaches the point where the proof script is used. Therefore, the static guarantees that this approach provides are severely limited by the fact that the proof scripts inside the tactics cannot be checked statically, when the tactic is defined.

Static proof scripts. This is the second fundamental issue we address in this paper. We show that the same staging construct utilized for introducing the extensible conversion rule, can be leveraged to perform *static proof checking for tactics*. The crucial point of our approach is the proof of existence of a transformation between proof objects, which suggests that under reasonable conditions, a proof script contained within a tactic can be transformed into a static proof script. This static script can then be evaluated at tactic definition time, to be checked for validity.

Last, we will show that this approach lends itself well to writing extensions of the conversion rule. We show that we can create a layering of conversion rules: using a basic conversion rule as a starting point, we can utilize it inside static proof scripts to implicitly prove the required obligations of a more advanced version, and so on. This minimizes the required user effort for writing new conversion rules, and enables truly modular proof checking.

3. Our toolbox

In this section, we will present the essential ingredients that are needed for the rest of our development. The main requirement is a language that supports type-safe manipulation of terms of a particular logic, as well as a general-purpose programming model that includes general recursion and other side-effectful operations. Two recently proposed languages for manipulating LF terms, Beluga [Pientka and Dunfield 2008] and Delphin [Poswolsky and Schürmann 2008], fit this requirement, as does VeriML [Stampoulis and Shao 2010], which is a language used to write type-safe tactics. Our discussion is focused on the latter, as it supports a richer ML-style calculus compared to the others, something useful for our purposes. Still, our results apply to all three.

We will now briefly describe the constructs that these languages support, as well as some new extensions that we propose. The interested reader can read more about these constructs in Sec. 6 and in our technical report [Stampoulis and Shao 2012].

A formal logic. The computational language we are presenting is centered around manipulation of terms of a specific formal logic. We will see more details about this logic in Sec. 4. For the time being, it will suffice to present a set of assumptions about the syntactic classes and typing judgements of this logic, shown in Fig. 3. Logical terms are represented by the syntactic class t, and include proof objects, propositions, terms corresponding to the domain of discourse (e.g. natural numbers), and the needed sorts and type constructors to classify such terms. Their variables are assigned types through an ordered context Φ. A package of a logical term t together with the variables context it inhabits Φ is called a contextual term and denoted as $T = [\Phi]\, t$. Our computational language works over contextual terms for reasons that will be evident later. The logic incorporates such terms by allowing them to get substituted for *meta-variables* X, using the constructor X/σ. When a term $T = [\Phi']\, t$ gets substituted for X, we go from the Φ' context to the current context Φ using the substitution σ.

Logical terms are classified using other logical terms, based on the normal variables environment Φ, and also an environment Ψ that types meta-variables, thus leading to the $\Psi;\, \Phi \vdash t : t'$ judgement. For example, a term t representing a closed proposition will be typed as $\bullet;\, \bullet \vdash t : \mathsf{Prop}$, while a proof object t_{pf} proving that proposition will satisfy the judgement $\bullet;\, \bullet \vdash t_{\text{pf}} : t$.

ML-style functional programming. We move on to the computational language. As its main core, we assume an ML-style functional language, supporting general recursion, algebraic data types and mutable references (see Fig. 4). Terms of this fragment are typed under a computational variables environment Γ and a store typing environment Σ, mapping mutable locations to types. Typing judgements are entirely standard, leading to a $\Sigma;\, \Gamma \vdash e : \tau$ judgement for typing expressions.

Dependently-typed programming over logical terms. As shown in Fig. 5, the first important additions to the ML computational core are constructs for dependent functions and products over contextual terms T. Abstraction over contextual terms is denoted as $\lambda X : T.e$. It has the dependent function type $(X : T) \to \tau$. The type is dependent since the introduced logical term might be used as the type of

$$k ::= * \mid k_1 \to k_2$$
$$\tau ::= \mathsf{unit} \mid \mathsf{int} \mid \mathsf{bool} \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \mu\alpha : k.\tau$$
$$\mid \forall\alpha : k.\tau \mid \alpha \mid \mathsf{array}\ \tau \mid \lambda\alpha : k.\tau \mid \tau_1\ \tau_2 \mid \cdots$$
$$e ::= () \mid n \mid e_1 + e_2 \mid e_1 \leq e_2 \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else} e_2$$
$$\mid \lambda\mathbf{x} : \tau.e \mid e_1\ e_2 \mid (e_1,\ e_2) \mid \mathsf{proj}_i\ e \mid \mathsf{inj}_i\ e$$
$$\mid \mathsf{case}(e, \mathbf{x}_1.e_1, \mathbf{x}_2.e_2) \mid \mathsf{fold}\ e \mid \mathsf{unfold}\ e \mid \Lambda\alpha : k.e \mid e\ \tau$$
$$\mid \mathsf{fix}\ \mathbf{x} : \tau.e \mid \mathsf{mkarray}(e, e') \mid e[e'] \mid e[e'] := e'' \mid l \mid \mathsf{error} \mid \cdots$$
$$\Gamma ::= \bullet \mid \Gamma,\ \mathbf{x} : \tau \mid \Gamma,\ \alpha : k \qquad \Sigma ::= \bullet \mid \Sigma,\ l : \mathsf{array}\ \tau$$

Figure 4. Syntax for the computational language (ML fragment)

$$\tau\ ::= \cdots \mid (X : T) \to \tau \mid (X : T) \times \tau \mid (\phi : \mathsf{ctx}) \to \tau$$
$$e\ ::= \cdots \mid \lambda X : T.e \mid e\ T \mid \lambda\phi : \mathsf{ctx}.e \mid e\ \Phi \mid \langle T,\ e\rangle$$
$$\mid \mathsf{let}\ \langle X,\ \mathbf{x}\rangle = e\ \mathsf{in}\ e'$$
$$\mid \mathsf{holcase}\ T\ \mathsf{return}\ \tau\ \mathsf{of}\ (T_1 \mapsto e_1)\cdots(T_n \mapsto e_n)$$
$$\mid \mathsf{ctxcase}\ \Phi\ \mathsf{return}\ \tau\ \mathsf{of}\ (\Phi_1 \mapsto e_1)\cdots(\Phi_n \mapsto e_n)$$

Figure 5. Syntax for the computational language (logical term constructs)

another term. An example would be a function that receives a proposition plus a proof object for that proposition, with type: $(P : \mathsf{Prop}) \to (X : P) \to \tau$. Dependent products that package a contextual logical term with an expression are introduced through the $\langle T,\ e\rangle$ construct and eliminated using $\mathsf{let}\ \langle X,\ \mathbf{x}\rangle = e\ \mathsf{in}\ e'$; their type is denoted as $(X : T) \times \tau$. Especially for packages of proof objects with the unit type, we introduce the syntax $\mathsf{LT}(T)$.

Last, in order to be able to support functions that work over terms in any context, we introduce context polymorphism, through a similarly dependent function type over contexts. With these in mind, we can define a simple tactic that gets a packaged proof of a universally quantified formula, and an instantiation term, and returns a proof of the instantiated formula as follows:

$$\mathsf{instantiate} : (\phi : \mathsf{ctx},\ T : [\phi]\,\mathsf{Type},\ P : [\phi,\ x : T]\,\mathsf{Prop},\ a : [\phi]\,T) \to \mathsf{LT}([\phi]\,\forall x : T, P) \to \mathsf{LT}([\phi]\,P/[\mathsf{id}_\phi,\ a])$$
$$\mathsf{instantiate}\ \phi\ T\ P\ a\ \mathsf{pf}\ = \mathsf{let}\ \langle H\rangle\ = \mathsf{pf}\ \mathsf{in}\ \langle H\ a\rangle$$

From here on, we will omit details about contexts and substitutions in the interest of presentation.

Pattern matching over terms. The most important new construct that VeriML supports is a pattern matching construct over logical terms denoted as holcase. This construct is used for dependent matching of a logical term against a set of patterns. The return clause specifies its return type; we omit it when it is easy to infer. Patterns are normal terms that include unification variables, which can be present under binders. This is the essential reason why contextual terms are needed.

Pattern matching over environments. For the purposes of our development, it is very useful to support one more pattern matching construct: matching over logical variable contexts. When trying to construct a certain proof, the logical environment represents what the current proof context is: what the current logical hypotheses at hand are, what types of terms have been quantified over, etc. By being able to pattern match over the environment, we can "look up" things in our current set of hypotheses, in order to prove further propositions. We can thus view the current environment as representing a simple form of the current *proof state*; the pattern matching construct enables us to manipulate it in a type-safe manner.

One example is an "assumption" tactic, that tries to prove a proposition by searching for a matching hypotheses in the context:

$$\mathsf{assumption} : (\phi : \mathsf{ctx}, P : \mathsf{Prop}) \to \mathsf{option}\ \mathsf{LT}(P)$$
$$\mathsf{assumption}\ \phi\ P =$$
$$\mathsf{ctxcase}\ \phi\ \mathsf{of}$$
$$\phi',\ H : P \mapsto \mathsf{return}\ \langle H\rangle$$
$$\mid \phi',\ \_ \quad \mapsto \mathsf{assumption}\ \phi'\ P$$

Proof object erasure semantics (new feature). The only construct that can influence the evaluation of a program based on the structure of a logical term is the pattern matching construct. For our purposes, pattern matching on proof objects is not necessary – we never look into the structure of a completed proof. Thus we can have the typing rules of the pattern matching construct specifically disallow matching on proof objects.

In that case, we can define an alternate operational semantics for our language where all proof objects are *erased* before using the original small-step reduction rules. Because of type safety, these proof-erasure semantics are guaranteed to yield equivalent results: even if no proof objects are generated, they are still bound to exist.

Implicit arguments. Let us consider again the *instantiate* function defined earlier. This function expects five arguments. From its type alone, it is evident that only the last two arguments are strictly necessary. The last argument, corresponding to a proof expression for the proposition $\forall x : T, P$, can be used to reconstruct exactly the arguments ϕ, T and P. Furthermore, if we know what the resulting type of a call to the function needs to be, we can choose even the instantiation argument a appropriately. We employ a simple inferrence mechanism so that such arguments are omitted from our programs. This feature is also crucial in our development in order to implicitly maintain and utilize the current proof state within our proof scripts.

Minimal staging support (new feature). Using the language we have seen so far we are able to write powerful tactics using a general-purpose programming model. But what if, inside our programs, we have calls to tactics where all of their arguments are constant? Presumably, those tactic calls could be evaluated to proof objects prior to tactic invocation. We could think of this as a form of generalized constant folding, which has one intriguing benefit: we can tell statically whether the tactic calls succeed or not.

This paper is exactly about exploring this possibility. Towards this effect, we introduce a rudimentary staging construct in our computational language. This takes the form of a letstatic construct, which binds a static expression to a variable. The static expression is evaluated during stage one (see Fig. 2), and can only depend on other static expressions. Details of this construct are presented in Fig. 11d and also in Sec. 6. After this addition, expressions in our language have a three-phase lifetime, that are also shown in Fig. 2.

- type-checking, where the well-formedness of expressions according to the rules of the language is checked, and inference of implicit arguments is performed
- static evaluation, where expressions inside letstatic are reduced to values, yielding a residual expression
- run-time, where the residual expression is evaluated

4. Extensible conversion rule

With these tools at hand, let us now return to the first issue that motivates us: the fact that proof checking is rigid and cannot be extended with user-defined procedures. As we have said in our introduction, many modern proof assistants are based on logics that include a *conversion rule*. This rule essentially identifies proposi-

$$
\begin{array}{rrcl}
\text{(sorts)} & s & ::= & \mathsf{Type} \mid \mathsf{Type}' \\
\text{(kinds)} & \mathcal{K} & ::= & \mathsf{Prop} \mid \mathsf{Nat} \mid \mathcal{K}_1 \to \mathcal{K}_2 \\
\text{(props.)} & P & ::= & P_1 \to P_2 \mid \forall x : \mathcal{K}.P \mid x \mid \mathsf{True} \\
 & & & \mid \mathsf{False} \mid P_1 \wedge P_2 \mid \cdots \\
\text{(dom.obj.)} & d & ::= & \mathsf{Zero} \mid \mathsf{Succ}\, d \mid P \mid \cdots \\
\text{(proof objects)} & \pi & ::= & x \mid \lambda x : P.\pi \mid \pi_1\ \pi_2 \mid \lambda x : \mathcal{K}.\pi \\
 & & & \mid \pi\ d \mid \cdots \\
\text{(HOL terms)} & t & ::= & s \mid \mathcal{K} \mid P \mid d \mid \pi
\end{array}
$$

Selected rules:

$$
\to\textsc{Intro}\ \frac{\Psi;\ \Phi, x : P \vdash \pi : P'}{\Psi;\ \Phi \vdash \lambda x : P.\pi : P \to P'}
\qquad
\to\textsc{Elim}\ \frac{\Psi;\ \Phi \vdash \pi : P \to P' \quad \Psi;\ \Phi \vdash \pi' : P}{\Psi;\ \Phi \vdash \pi\ \pi' : P'}
$$

Figure 6. Syntax and selected rules of the logic language λHOL

$$
\textsc{Conversion}\quad \frac{\Psi;\ \Phi \vdash_c \pi : P \qquad P =_{\beta\mathbb{N}} P'}{\Psi;\ \Phi \vdash_c \pi : P'}
$$

$\boxed{d \to_{\beta\mathbb{N}} d'}$

$$
\begin{array}{l}
(\lambda x : \mathcal{K}.d)\ d' \to_{\beta\mathbb{N}} d[d'/x] \\
\mathsf{natElim}_{\mathcal{K}}\ d_z\ d_s\ \mathsf{zero} \to_{\beta\mathbb{N}} d_z \\
\mathsf{natElim}_{\mathcal{K}}\ d_z\ d_s\ (\mathsf{succ}\ d) \to_{\beta\mathbb{N}} d_s\ d\ (\mathsf{natElim}_{\mathcal{K}}\ d_z\ d_s\ d)
\end{array}
$$

$\boxed{d =_{\beta\mathbb{N}} d'}$ is the compatible, reflexive, symmetric and transitive closure of $d \to_{\beta\mathbb{N}} d'$

Figure 7. Extending λHOL with the conversion rule ($\lambda\mathrm{HOL}_c$)

tions up to some equivalence relation: usually this is equivalence up to partial evaluation of the functions contained within propositions.

The supported relation is decided when the logic is designed. Any extension to this relation requires a significant amount of work, both in terms of implementation, and in terms of metatheoretic proof required. This is evidenced by projects that extend the conversion rule in Coq, such as Blanqui et al. [1999] and Strub [2010]. Even if user extensions are supported, those only take the form of first-order theories. Can we do better than this, enabling arbitrarily complex user extensions, written with the full power of ML, yet maintaining soundness?

It turns out that we can: this is the subject of this section. The key idea is to recognize that the conversion rule is essentially a tactic, embedded within the type checker of the logic. Calls to this tactic are made implicitly as part of checking a given proof object for validity. So how can we support a flexible, extensible alternative? Instead of hardcoding a conversion tactic within the logic type checker, we can program a type-safe version of the same tactic within VeriML, with the requirement that it provides proof of the claimed equivalence. Instead of calling the conversion tactic as part of proof checking, we use staging to call the tactic statically – after (VeriML) type checking, but before runtime execution. This can be viewed as a second, potentially non-terminating proof checking stage. Users are now free to write their own conversion tactics, extending the static checking available for proof objects and proof scripts. Still, soundness is maintained, since full proof objects in the original logic can always be constructed. As an example, we have extended the conversion rule that we use by a congruence closure procedure, which makes use of mutable data structures, and by an arithmetic simplification procedure.

4.1 Introducing: the conversion rule

First, let us present what the conversion rule really is in more detail. We will base our discussion on a simple type-theoretic higher-order

$$
\frac{\Psi;\ \Phi \vdash_e d_1 : \mathcal{K} \qquad \Psi;\ \Phi \vdash_c d_2 : \mathcal{K}}{\Psi;\ \Phi \vdash_e d_1 = d_2 : \mathsf{Prop}}
\qquad
\frac{\Psi;\ \Phi \vdash_e d : \mathcal{K}}{\Psi;\ \Phi \vdash_e \mathsf{refl}\ d : d = d}
$$

$$
\frac{\Psi;\ \Phi,\ x : \mathcal{K} \vdash_e P : \mathsf{Prop} \qquad \Psi;\ \Phi \vdash_e d_1 : \mathcal{K} \qquad \Psi;\ \Phi \vdash_e \pi : P[d_1/x] \qquad \Psi;\ \Phi \vdash_e \pi' : d_1 = d_2}{\Psi;\ \Phi \vdash_e \mathsf{leibniz}\ (\lambda x : \mathcal{K}.P)\ \pi\ \pi' : P[d_2/x]}
$$

$$
\frac{\Psi;\ \Phi,\ x : \mathcal{K} \vdash_e \pi : d_1 = d_2}{\Psi;\ \Phi \vdash_e \mathsf{lamEq}\ (\lambda x : \mathcal{K}.\pi) : (\lambda x : \mathcal{K}.d_1) = (\lambda x : \mathcal{K}.d_2)}
$$

$$
\frac{\Psi;\ \Phi,\ x : \mathcal{K} \vdash_e \pi : d_1 = d_2 \qquad \Psi;\ \Phi \vdash_e d_1 : \mathsf{Prop}}{\Psi;\ \Phi \vdash_e \mathsf{forallEq}\ (\lambda x : \mathcal{K}.\pi) : (\forall x : \mathcal{K}.d_1) = (\forall x : \mathcal{K}.d_2)}
$$

$$
\frac{\Psi;\ \Phi,\ x : \mathcal{K} \vdash_e d : \mathcal{K}' \qquad \Psi;\ \Phi \vdash_e d' : \mathcal{K}}{\Psi;\ \Phi \vdash_e \mathsf{betaEq}\ (\lambda x : \mathcal{K}.d)\ d' : (\lambda x : \mathcal{K}.d)\ d' = d[d'/x]}
$$

Axioms assumed:

$$
\begin{array}{lcl}
\mathsf{natElimBase}_{\mathcal{K}} & : & \forall f_z.\forall f_s.\mathsf{natElim}_{\mathcal{K}}\ f_z\ f_s\ \mathsf{zero} = f_z \\
\mathsf{natElimStep}_{\mathcal{K}} & : & \forall f_z.\forall f_s.\forall n.\ \mathsf{natElim}_{\mathcal{K}}\ f_z\ f_s\ (\mathsf{succ}\ n) = \\
 & & \qquad f_s\ n\ (\mathsf{natElim}_{\mathcal{K}}\ f_z\ f_s\ n)
\end{array}
$$

Figure 8. Extending λHOL with explicit equality ($\lambda\mathrm{HOL}_e$)

logic, based on the λHOL logic as described in Barendregt and Geuvers [1999], and used in our original work on VeriML [Stampoulis and Shao 2010]. We can think of such a logic composed by the following broad classes: the objects of the domain of discourse d, which are the objects that the logic reasons about, such as natural numbers and lists; their classifiers, the kinds $\mathcal{K}$ (classified in turn by sorts s); the propositions P; and the derivations, which prove that a certain proposition is true. We can represent derivations in a linear form as terms π in a typed lambda-calculus; we call such terms proof objects, and their types represent propositions in the logic. Checking whether a derivation is a valid proof of a certain proposition amounts to type-checking its corresponding proof object. Some details of this logic are presented in Fig. 6; the interested reader can find more information about it in the above references and in our technical report [Stampoulis and Shao 2012].

In Fig. 6, we show what the conversion rule looks like for this logic: it is a typing judgement that effectively identifies propositions up to an equivalence relation, with respect to checking proof objects. We call this version of the logic $\lambda\mathrm{HOL}_c$ and use $\vdash_c$ to denote its entailment relation. The equivalence relation we consider in the conversion rule is evaluation up to β-reductions and uses of primitive recursion of natural numbers, denoted as natElim. In this way, trivial arguments based on this notion of computation alone need not be witnessed, as for example is the fact that $(\mathsf{Succ}\ x) + y = \mathsf{Succ}\ (x + y)$ – when the addition function is defined by primitive recursion on the first argument. Of course, this is only a very basic use of the conversion rule. It is possible to omit larger proofs through much more sophisticated uses. This leads to simpler proofs and smaller proof objects.

Still, when using this approach, the choice of what relation is supported by the conversion rule needs to be made during the definition of the logic. This choice permeates all aspects of the metatheory of the logic. It is easy to see why, even with the tiny fragment of logic we have introduced. Most typing rules for proof objects in the logic are similar to the rules →Intro and →Elim: they are syntax-directed. This means that upon seeing the associated proof object constructor, like $\lambda x : P.\pi$ in the case of →Intro, we can directly tell that it applies. If all rules were syntax directed, it would

```
βℕequal : (ϕ : ctx, T : Type, t1 : T, t2 : T) → option LT(t1 = t2)
βℕequal ϕ T t1 t2 =
  holcase whnf ϕ T t1, whnf ϕ T t2 of
     ((ta : T′ → T) tb), (tc td) ↦
        do ⟨pf1⟩   ←   βℕequal ϕ (T′ → T) ta tc
           ⟨pf1⟩   ←   βℕequal ϕ T′ tb td
           return ⟨··· proof of ta tb = tc td ···⟩
     | (ta → tb), (tc → td) ↦
        do ⟨pf1⟩   ←   βℕequal ϕ Prop ta tc
           ⟨pf1⟩   ←   βℕequal ϕ Prop tb td
           return ⟨··· proof of ta → tb = tc → td ···⟩
     | (λx : T.t1), (λx : T.t2) ↦
        do ⟨pf⟩   ←   βℕequal [ϕ, x : T] Prop t1 t2
           return ⟨··· proof of λx : T.t1 = λx : T.t2 ···⟩
     | t1, t1 ↦ do return ⟨··· proof of t1 = t1 ···⟩
     | t1, t2 ↦ None

requireEqual : (ϕ : ctx, T : Type, t1 : T, t2 : T).LT(t1 = t2)
requireEqual ϕ T t1 t2 =
  match βℕequal ϕ T t1 t2 with Some x ↦ x | None ↦ error
```

Figure 9. VeriML tactic for checking equality up to β-conversion

be entirely simple to prove that the logic is sound by an inductive argument: essentially, since no proof constructor for False exists, there is no valid derivation for False.

In this logic, the only rule that is not syntax directed is exactly the conversion rule. Therefore, in order to prove the soundness of the logic, we have to show that the conversion rule does not somehow introduce a proof of False. This means that proving the soundness of the logic passes essentially through the specific relation we have chosen for the conversion rule. Therefore, this approach is foundationally limited from supporting user extensions, since any new extension would require a new metatheoretic result in order to make sure that it does not violate logical soundness.

4.2 Throwing conversion away

Since having a fixed conversion rule is bound to fail if we want it to be extensible, what choice are we left with, but to throw it away? This radical sounding approach is what we will do here. We can replace the conversion rule by an explicit notion of equality, and provide explicit proof witnesses for rewriting based on that equality. Essentially, all the points where the conversion rule was alluded to and proofs were omitted, need now be replaced by proof objects witnessing the equivalence. Some details for the additions required to the base λHOL logic are shown in Fig. 8, yielding the λHOL_e logic. There are good reasons for choosing this version: first, the proof checker is as simple as possible, and does not need to include the conversion checking routine. We could view this routine as performing proof search over the replacement rules, so it necessarily is more complicated, especially since it needs to be relatively efficient. Also, the metatheory of the logic itself can be simplified. Even when the conversion rule is supported, the metatheory for the associated logic is proved through the explicit equality approach; this is because model construction for a logic benefits from using explicit equality [Siles and Herbelin 2010].

Still, this approach has a big disadvantage: the proof objects soon become extremely large, since they include painstakingly detailed proofs for even the simplest of equivalences. This precludes their use as independently checkable proof certificates that can be sent to a third party. It is possible that this is one of the reasons why systems based on logics with explicit equalities, such as HOL4

```
whnf : (ϕ : ctx, T : Type, t : T) → (t′ : T) × LT(t = t′)
whnf ϕ T t = holcase t of
   (t1 : T′ → T)(t2 : T′) ↦
      let ⟨t′1, pf1⟩ = whnf ϕ (T′ → T) t1 in
      holcase t′1 of
         λx : T′.tf   ↦   ⟨[ϕ] tf/[idΦ, t2], ···⟩
         | t′1        ↦   ⟨[ϕ] t′1 t2, ···⟩
   | natElimK fz fs n ↦
        let ⟨n′, pf1⟩ = whnf ϕ Nat n in holcase n′ of
              zero      ↦   ⟨[ϕ] fz, ···⟩
            | succ n′   ↦   ⟨[ϕ] fs n′ (natElimK fz fs n′), ···⟩
            | n′        ↦   ⟨[ϕ] natElimK fz fs n′, ···⟩
   | t ↦ ⟨t, ···⟩
```

Figure 10. VeriML tactic for rewriting to weak head-normal form

[Slind and Norrish 2008] and Isabelle/HOL [Nipkow et al. 2002], do not generate proof objects by default.

4.3 Getting conversion back

We will now see how it is possible to reconcile the explicit equality based approach with the conversion rule: we will gain the conversion rule back, albeit it will remain completely outside the logic. Therefore we will be free to extend it, all the while without risking introducing unsoundness in the logic, since the logic remains fixed (λHOL_e as presented above).

We do this by revisiting the view of the conversion rule as a special "trusted" tactic, through the tools presented in the previous section. First, instead of hardcoding a conversion tactic in the type checker, we program a *type-safe conversion tactic*, utilizing the features of VeriML. Based on typing alone we require that it returns a valid proof of the claimed equivalences:

$$\beta\mathbb{N}\mathsf{equal} : (\phi : \mathsf{ctx},\ T : \mathsf{Type},\ t : T,\ t' : T) \to \mathsf{option\ LT}(t = t')$$

Second, we evaluate this tactic under *proof erasure semantics*. This means that no proof objects are produced, leading to the same space gains as the original conversion rule. Third, we use the staging construct in order to *check conversion statically*.

Details. We now present our approach in more detail. First, in Fig. 9, we show a sketch of the code behind the type-safe conversion check tactic. It works by first rewriting its input terms into weak head-normal form, via the whnf function in Fig. 10, and then recursively checking their subterms for equality. In the equivalence checking function, more cases are needed to deal with quantification; while in the rewriting procedure, a recursive call is missing, which would complicate our presentation here. We also define a version of the tactic that raises an error instead of returning an option type if we fail to prove the terms equal, which we call requireEqual. The full details can be found in our implementation.

The code of the βℕequal tactic is in fact entirely similar to the code one would write for the conversion check routine inside a logic type checker, save for the extra types and proof objects. It therefore follows trivially that everything that holds for the standard implementation of the conversion check also holds for this code: e.g. it corresponds exactly to the $=_{\beta\mathbb{N}}$ relation as defined in the logic; it is bound to terminate because of the strong normalization theorem for this relation; and its proof-erased version is at least as trustworthy as the standard implementation.

Furthermore, given this code, we can produce a form of *typed proof scripts* inside VeriML that correspond exactly to proof objects in the logic with the conversion rule, both in terms of their actual code, and in terms of the steps required to validate them. This is done by constructing a proof script in VeriML by induction on the derivation of the proof object in λHOL_c, replacing each proof object constructor by an equivalent VeriML tactic as follows:

constructor	to tactic	of type
$\lambda x : P.\pi$	Assume e	$\mathsf{LT}([\phi,\ H : P]\,P') \to \mathsf{LT}(P \to P')$
$\pi_1\ \pi_2$	Apply $e_1\ e_2$	$\mathsf{LT}(P \to P') \to \mathsf{LT}(P) \to \mathsf{LT}(P')$
$\lambda x : \mathcal{K}.\pi$	Intro e	$\mathsf{LT}([\phi,\ x : T]\,P') \to \mathsf{LT}(\forall x : T, P')$
$\pi\ d$	Inst $e\ a$	$\mathsf{LT}(\forall x : T, P) \to (a : T) \to$ $\mathsf{LT}(P/[\mathsf{id},\ a])$
c	Lift c	$(H : P) \to \mathsf{LT}(P)$
(conversion)	Conversion	$\mathsf{LT}(P) \to \mathsf{LT}(P = P') \to \mathsf{LT}(P')$

Here we have omitted the current logical environment ϕ; it is maintained through syntactic means as discussed in Sec. 7 and through type inference. The only subtle case is conversion. Given the transformed proof e for the proof object π contained within a use of the conversion rule, we call the conversion tactic as follows:

$$\mathsf{letstatic\ pf}\ =\ \mathsf{requireEqual}\ P\ P'\ \mathsf{in\ Conversion\ e\ pf}$$

The arguments to requireEqual can be easily inferred, making crucial use of the rich type information available. Conversion could also be used implicitly in the other tactics. Thus the resulting expression looks entirely identical to the original proof object.

Correspondence with original proof object. In order to elucidate the correspondence between the resulting proof script expression and the original proof object, it is fruitful to view the proof script as a proof certificate, sent to a third party. The steps required to check whether it constitutes a valid proof are the following. First, the whole expression is checked using the type checker of the computational language. Then, the calls to the requireEqual function are evaluated during stage one, using proof erasure semantics. We expect them to be successful, just as we would expect the conversion rule to be applicable when it is used. Last, the rest of the tactics are evaluated; by a simple argument, based on the fact that they do not use pattern matching or side-effects, they are guaranteed to terminate and produce a proof object in λHOL_e. This validity check is entirely equivalent to the behavior of type-checking the λHOL_c proof object, save for pushing all conversion checks towards the end.

4.4 Extending conversion at will

In our treatment of the conversion rule we have so far focused on regaining the $\beta\mathbb{N}$ conversion in our framework. Still, there is nothing confining us to supporting this conversion check only. As long as we can program a conversion tactic in VeriML that has the right type, it can safely be made part of our conversion rule.

For example, we have written an eufEqual function, which checks terms for equivalence based on the equality with uninterpreted functions decision procedure. It is adapted from our previous work on VeriML [Stampoulis and Shao 2010]. This equivalence checking tactic isolates hypotheses of the form $d_1 = d_2$ from the current context, using the newly-introduced context matching support. Then, it constructs a union-find data structure in order to form equivalence classes of terms. Based on this structure, and using code similar to $\beta\mathbb{N}$equal (recursive calls on subterms), we can decide whether two terms are equal up to simple uses of the equality hypotheses at hand. We have combined this tactic with the original $\beta\mathbb{N}$equal tactic, making the implicit equivalence supported similar to the one in the Calculus of Congruent Constructions [Blanqui et al. 2005]. This demonstrates the flexibility of this approach: equivalence checking is extended with a sophisticated decision procedure, which is programmed using its original, imperative formulation. We have programmed both the rewriting procedure and the equality checking procedure in an extensible manner, so that we can globally register further extensions.

4.5 Typed proof scripts as certificates

Earlier we discussed how we can validate the proof scripts resulting from turning the conversion rule into explicit tactic calls. This discussion shows an interesting aspect of typed proof scripts: they can be viewed as a proof witness that is a flexible compromise between untyped proof scripts and proof objects. When a typed proof script consists only of static calls to conversion tactics and uses of total tactics, it can be thought of as a proof object in a logic with the corresponding conversion rule. When it also contains other tactics, that perform potentially expensive proof search, it corresponds more closely to an untyped proof script, since it needs to be fully evaluated. Still, we are allowed to validate parts of it statically. This is especially useful when developing the proof script, because we can avoid the evaluation of expensive tactic calls while we focus on getting the skeleton of the proof correct.

Using proof erasure for evaluating requireEqual is only one of the choices the receiver of such a proof certificate can make. Another choice would be to have the function return an actual proof object, which we can check using the λHOL_e type checker. In that case, the VeriML interpreter does not need to become part of the trusted base of the system. Last, the ‘safest possible’ choice would be to avoid doing any evaluation of the function, and ask the proof certificate provider to do the evaluation of requireEqual themselves. In that case, no evaluation of computational code would need to happen at the proof certificate receiver’s side. This mitigates any concerns one might have for code execution as part of proof validity checking, and guarantees that the small λHOL_e type checker is the trusted base in its entirety. Also, the receiver can decide on the above choices selectively for different conversion tactics – e.g. use proof erasure for $\beta\mathbb{N}$equal but not for eufEqual, leading to a trusted base identical to the λHOL_c case. This means that the choice of the conversion rule rests with the proof certificate receiver and not with the designer of the logic. Thus the proof certificate receiver can choose the level of trust they require at will.

5. Static proof scripts

In the previous section, we have demonstrated how proof checking for typed proof scripts can be made user-extensible, through a new treatment of the conversion rule. It makes use of user-defined, type-safe tactics, which are evaluated statically. The question that remains is what happens with respect to proofs within tactics. If a proof script is found within a tactic, must we wait until that evaluation point is reached to know whether the proof script is correct or not? Or is there a way to check this statically, as soon as the tactic is defined?

In this section we show how this is possible to do in VeriML using the staging construct we have introduced. Still, in this case matters are not as simple as evaluating certain expressions statically rather than dynamically. The reason is that proof scripts contained within tactics mention uninstantiated meta-variables, and thus cannot be evaluated through staging. We resolve this by showing the existence of a transformation, which “collapses” logical terms from an arbitrary meta-variables context into the empty one.

We will focus on the case of developing conversion routines, similar to the ones we saw earlier. The ideas we present are generally applicable when writing other types of tactics as well; we focus on conversion routines in order to demonstrate that the two main ideas we present in this paper can work in tandem.

A rewriter for plus. We will consider the case of writing a rewriter –similar to whnf– for simplifying expressions of the form $x+y$, depending on the second argument. The addition function is defined by induction on the first argument, as follows:

$$(+) = \lambda x.\lambda y.\mathsf{natElim}_{\mathsf{Nat}}\ y\ (\lambda p.\lambda r.\mathsf{Succ}\ r)\ x$$

In order for rewriters to be able to use existing as well as future rewriters to perform their recursive calls, we write them in the open recursion style – they receive a function of the same type that corresponds to the "current" rewriter. The code looks as follows:

$\mathsf{rewriterType} = (\phi : \mathsf{ctx}, T : \mathsf{Type}, t : T) \to (t' : T) \times \mathsf{LT}(t = t')$
$\mathsf{plusRewriter1} : \mathsf{rewriterType} \to \mathsf{rewriterType}$
$\mathsf{plusRewriter1}\ \mathsf{recursive}\ \phi\ T\ t = \mathsf{holcase}\ t\ \mathsf{with}$
$\quad x + y \mapsto$
$\qquad \mathsf{let}\ \langle y',\ \langle \mathsf{pfy}' \rangle\rangle = \mathsf{recursive}\ \phi\ y\ \mathsf{in}$
$\qquad \mathsf{let}\ \langle t',\ \langle \mathsf{pft}' \rangle\rangle =$
$\qquad\quad \mathsf{holcase}\ y'\ \mathsf{return}\ \Sigma t' : [\phi]\,\mathsf{Nat}.\mathsf{LT}([\phi]\,x + y' = t')\ \mathsf{of}$
$\qquad\quad\ 0 \quad\ \mapsto \langle x, \cdots\ \textit{proof of}\ \ x + 0 = x \cdots \rangle$
$\qquad\quad \mid \mathsf{Succ}\ y' \mapsto \Big\langle \mathsf{Succ}(x + y'),$
$\qquad\qquad\quad \cdots\ \textit{proof of}\ \ x + \mathsf{Succ}\ y' = \mathsf{Succ}\ (x + y') \cdots \Big\rangle$
$\qquad\quad \mid y' \quad\ \mapsto \langle x + y', \cdots\ \textit{proof of}\ \ x + y' = x + y' \cdots \rangle$
$\qquad \mathsf{in}\ \langle t',\ \langle \cdots\ \textit{proof of}\ \ x + y = t' \cdots \rangle\rangle$
$\quad \mid t \quad \mapsto \langle t, \cdots\ \textit{proof of}\ \ t = t \cdots \rangle$

While developing such a tactic, we can leverage the VeriML type checker to know the types of missing proofs. But how do we fill them in? For the interesting cases of $x + 0 = x$ and $x + \mathsf{Succ}\ y' = \mathsf{Succ}\ (x + y')$, we would certainly need to prove the corresponding lemmas. But for the rest of the cases, the corresponding lemmas would be uninteresting and tedious to state, such as the following for the $x + y = t'$ case:

$$\mathsf{lemma1} : \forall x, y, y', t', y = y' \to (x + y' = t') \to x + y = t$$

Stating and proving such lemmas soon becomes a hindrance when writing tactics. An alternative is to use the congruence closure conversion rule to solve this trivial obligation for us directly at the point where it is required. Our first attempt would be:

$\text{proof of}\ x + y = t' \ \equiv$
$\ \mathsf{let}\ \langle \mathsf{pf} \rangle = \mathsf{requireEqual}\ \ [\phi, H_1 : y = y', H_2 : x + y' = t']\ \ (x + y)\ \ t'$
$\ \mathsf{in}\ \big\langle [\phi]\,\mathsf{pf}/[\mathsf{id}_\phi,\ \mathsf{pfy}',\ \mathsf{pft'}] \big\rangle$

The benefit of this approach is evident when utilizing implicit arguments, since most of the details can be inferred and therefore omitted. Here we had to alter the environment passed to requireEqual, which includes several extra hypotheses. Once the resulting proof has been computed, the hypotheses are substituted by the actual proofs that we have.

The problem with this approach is two-fold: first, the call to the requireEqual tactic is recomputed every time we reach that point of our function. For such a simple tactic call, this does not impact the runtime significantly; still, if we could avoid it, we would be able use more sophisticated and expensive tactics. The second problem is that if for some reason the requireEqual is not able to prove what it is supposed to, we will not know until we actually reach that point in the function.

Moving to static proofs. This is where using the letstatic construct becomes essential. We can evaluate the call to requireEqual statically, during stage one interpretation. Thus we will know at the time that plusRewriter1 is defined whether the call succeeded; also, it will be replaced by a concrete value, so it will not affect the runtime behavior of each invocation of plusRewriter1 anymore. To do that, we need to avoid mentioning any of the metavariables that are bound during runtime, like x, y, and t'. This is done by specifying an appropriate environment in the call to requireEqual, similarly to the way we incorporated the extra knowledge above and substituted it later. Using this approach, we have:

$\text{proof of}\ x + y = t' \ \equiv$
$\quad \mathsf{letstatic}\ \langle \mathsf{pf} \rangle \ =$
$\qquad \mathsf{let}\ \phi' \ = [x, y, y', t' : \mathsf{Nat}, H_1 : y = y', H_2 : x + y' = t']\ \mathsf{in}$
$\qquad \mathsf{requireEqual}\ \phi'\ (x + y)\ t'$
$\quad \mathsf{in}\ \big\langle [\phi]\,\mathsf{pf}/[x/\mathsf{id}_\phi, y/\mathsf{id}_\phi, y'/\mathsf{id}_\phi, t'/\mathsf{id}_\phi, \mathsf{pfy}'/\mathsf{id}_\phi, \mathsf{pft}'/\mathsf{id}_\phi] \big\rangle$

What we are essentially doing here is replacing the meta-variables by normal logical variables, which our tactics can deal with. The meta-variable context is "collapsed" into a normal context; proofs are constructed using tactics in this environment; last, the resulting proofs are transported back into the desired context by substituting meta-variables for variables. We have explicitly stated the substitutions in order to distinguish between normal logical variables and meta-variables.

The reason why this transformation needs to be done is that functions in our computational language can only manipulate logical terms that are open with respect to a normal variables context; not logical terms that are open with respect to the meta-variables context too. A much more complicated, but also more flexible alternative to using this "collapsing" trick would be to support meta-n-variables within our computational language directly.

Overall, this approach is entirely similar to proving the auxiliary lemma mentioned above, prior to the tactic definition. The benefit is that by leveraging the type information together with type inference, we can avoid stating such lemmas explicitly, while retaining the same runtime behavior. We thus end up with very concise proof expressions that are statically validated. We introduce syntactic sugar for binding a static proof script to a variable, and then performing a substitution to bring it into the current context, since this is a common operation.

$$\langle e \rangle_{\mathrm{static}} \ \equiv\ \mathsf{letstatic}\ \langle \mathsf{pf} \rangle \ = \ e\ \mathsf{in}\ \langle [\phi]\,\mathsf{pf}/\cdots \rangle$$

Based on these, the trivial proofs in the above tactic can be filled in using a simple $\langle \mathsf{requireEqual} \rangle_{\mathrm{static}}$ call; for the other two we use $\langle \mathsf{Instantiate}\ (\mathsf{NatInduction}\ \mathsf{requireEqual}\ \mathsf{requireEqual})\ x \rangle_{\mathrm{static}}$.

After we define plusRewriter1, we can register it with the global equivalence checking procedure. Thus, all later calls to requireEqual will benefit from this simplification. It is then simple to prove commutativity for addition:

$\mathsf{plusComm} \quad : \quad \mathsf{LT}(\forall x, y. x + y = y + x)$
$\mathsf{plusComm} \quad = \quad \mathsf{NatInduction}\ \mathsf{requireEqual}\ \mathsf{requireEqual}$

Based on this proof, we can write a rewriter that takes commutativity into account and uses the hash values of logical terms to avoid infinite loops. We have worked on an arithmetic simplification rewriter that is built by layering such rewriters together, using previous ones to aid us in constructing the proofs required in later ones. It works by converting expressions into a list of monomials, sorting the list based on the hash values of the variables, and then factoring monomials on the same variable. Also, the eufEqual procedure mentioned earlier has all of its associated proofs automated through static proof scripts, using a naive, potentially non-terminating, equality rewriter.

Is collapsing always possible? A natural question to ask is whether collapsing the metavariables context into a normal context is always possible. In order to cast this as a more formal question, we notice that the essential step is replacing a proof object π of type $[\Phi]\,t$, typed under the meta-variables environment Ψ, by a proof object π' of type $[\Phi']\,t'$ typed under the empty meta-variables environment. There needs to be a substitution so that π' gets transported back to the Φ, Ψ environment, and has the appropriate type.

Syntax of the logic

$$(\textit{terms})\; t ::= s \mid c \mid f_i \mid b_i \mid \lambda(t_1).t_2 \mid t_1\ t_2 \mid \Pi(t_1).t_2 \mid t_1 = t_2 \mid \mathsf{refl}\ t \mid \mathsf{leibniz}\ t_1\ t_2 \mid \mathsf{lamEq}\ t \mid \mathsf{forallEq}\ t_1\ t_2 \mid \mathsf{betaEq}\ t_1\ t_2$$

$$(\textit{sorts})\; s ::= \mathsf{Prop} \mid \mathsf{Type} \mid \mathsf{Type}' \qquad (\textit{var. context})\quad \Phi ::= \bullet \mid \Phi,\, t \qquad (\textit{substitutions})\quad \sigma ::= \bullet \mid \sigma,\, t$$

Example of representation: $a : \mathsf{Nat} \vdash \lambda x : \mathsf{Nat}.(\lambda y : \mathsf{Nat}.\mathsf{refl}\ (\mathsf{plus}\ a\ y))(\mathsf{plus}\ a\ x) \;\mapsto\; \mathsf{Nat} \vdash \lambda(\mathsf{Nat}).(\lambda(\mathsf{Nat}).\mathsf{refl}\ (\mathsf{plus}\ f_0\ b_0))\ (\mathsf{plus}\ f_0\ b_0)$

Freshen: $\lceil t \rceil^n_m$

$$\begin{array}{lcl} \lceil f_i \rceil & = & f_i \\ \lceil b_n \rceil^n_m & = & f_m \\ \lceil b_i \rceil^n & = & b_i \text{ when } i < n \\ \lceil (\lambda(t_1).t_2) \rceil^n & = & \lambda(\lceil t_1 \rceil^n).\, \lceil t_2 \rceil^{n+1} \\ \lceil t_1\ t_2 \rceil & = & \lceil t_1 \rceil\ \lceil t_2 \rceil \end{array}$$

Bind: $\lfloor t \rfloor^n_m$

$$\begin{array}{lcl} \lfloor f_{m-1} \rfloor^n_m & = & b_n \\ \lfloor f_i \rfloor^n_m & = & f_i \text{ when } i < m-1 \\ \lfloor b_i \rfloor & = & b_{i+1} \\ \lfloor (\lambda(t_1).t_2) \rfloor & = & \lambda(\lfloor t_1 \rfloor^n).\, \lfloor t_2 \rfloor^{n+1} \\ \lfloor t_1\ t_2 \rfloor & = & \lfloor t_1 \rfloor\ \lfloor t_2 \rfloor \end{array}$$

(a) Hybrid deBruijn levels-deBruijn indices representation technique

---

Syntax

$$t ::= \cdots \mid f_{\mathbf{I}} \mid X_i/\sigma \quad \Phi ::= \bullet \mid \Phi,\, t \mid \Phi,\, \phi_i \quad \sigma ::= \bullet \mid \sigma,\, t \mid \sigma,\, \mathbf{id}(\phi_i) \quad (\textit{indices})\ \mathbf{I} ::= n \mid \mathbf{I} + |\phi_i| \quad (\textit{ctx.terms})\ T ::= [\Phi]\,t \mid [\Phi]\,\Phi'$$

$$(\textit{ctx.kinds})\ K ::= [\Phi]\,t \mid [\Phi]\,\mathsf{ctx} \qquad (\textit{extension context})\ \Psi ::= \bullet \mid \Psi,\, K \qquad (\textit{ext. subst.})\ \sigma_\Psi ::= \bullet \mid \sigma_\Psi,\, T$$

$\Psi;\, \Phi \vdash t : t'$ (sample)

$$\frac{\Phi.\mathbf{I} = t}{\Psi;\, \Phi \vdash f_{\mathbf{I}} : t} \qquad \frac{\Psi;\, \Phi \vdash t_1 : \Pi(t).t' \qquad \Psi;\, \Phi \vdash t_2 : t}{\Psi;\, \Phi \vdash t_1\ t_2 : \lceil t' \rceil \cdot (\mathsf{id}_\Phi, t_2)} \qquad \frac{\Psi.i = [\Phi']\,t' \qquad \Psi;\, \Phi \vdash \sigma : \Phi'}{\Psi;\, \Phi \vdash X_i/\sigma : t' \cdot \sigma}$$

$\Psi \vdash T : K$

$$\frac{\Psi;\, \Phi \vdash t : t'}{\Psi \vdash [\Phi]\,t : [\Phi]\,t'} \qquad \frac{\Psi \vdash \Phi,\, \Phi' \text{ wf}}{\Psi \vdash [\Phi]\,\Phi' : [\Phi]\,\mathsf{ctx}}$$

$\Psi \vdash \Phi$ wf (sample)

$$\frac{\Psi \vdash \Phi \text{ wf} \qquad \Psi.i = [\Phi]\,\mathsf{ctx}}{\Psi \vdash (\Phi,\, \phi_i) \text{ wf}}$$

(b) Extension variables: meta-variables and context variables

---

Subst. application: $t \cdot \sigma$

$$c \cdot \sigma = c \qquad f_{\mathbf{I}} \cdot \sigma = \sigma.\mathbf{I} \qquad b_i \cdot \sigma = b_i \qquad (\lambda(t_1).t_2) \cdot \sigma = \lambda(t_1 \cdot \sigma).(t_2 \cdot \sigma) \qquad (t_1\ t_2) \cdot \sigma = (t_1 \cdot \sigma)\ (t_2 \cdot \sigma)$$

Ext. subst. application (sample)

$$(\mathbf{I},\, |\phi_i|) \cdot \sigma_\Psi = (\mathbf{I} \cdot \sigma_\Psi),\ |\Phi'| \text{ when } \sigma_\Psi.i = [\_]\,\Phi' \qquad (X_i/\sigma) \cdot \sigma_\Psi = t \cdot (\sigma \cdot \sigma_\Psi) \text{ when } \sigma_\Psi.i = [\_]\,t$$

$$(\sigma,\, \mathbf{id}(\phi_i)) \cdot \sigma_\Psi = \sigma \cdot \sigma_\Psi,\ \mathsf{id}_{\sigma_\Psi.i} \qquad (\Phi,\, \phi_i) \cdot \sigma_\Psi = \Phi \cdot \sigma_\Psi,\ \Phi' \text{ when } \sigma_\Psi.i = [\_]\,\Phi'$$

$\Psi;\, \Phi \vdash \sigma : \Phi'$

$$\frac{}{\Psi;\, \Phi \vdash \bullet : \bullet} \qquad \frac{\Psi;\, \Phi \vdash \sigma : \Phi' \qquad \Psi;\, \Phi \vdash t : t' \cdot \sigma}{\Psi;\, \Phi \vdash (\sigma,\, t) : (\Phi',\, t')} \qquad \frac{\Psi;\, \Phi \vdash \sigma : \Phi' \qquad \Psi.i = [\Phi']\,\mathsf{ctx} \qquad \Phi',\, \phi_i \subseteq \Phi}{\Psi;\, \Phi \vdash (\sigma,\, \mathbf{id}(\phi_i)) : (\Phi',\, \phi_i)}$$

$\Psi \vdash \sigma_\Psi : \Psi'$ (selected)

$$\frac{\Psi \vdash \sigma_\Psi : \Psi' \qquad \Psi \vdash T : K \cdot \sigma_\Psi}{\Psi \vdash (\sigma_\Psi,\, T) : (\Psi',\, K)}$$

Subst. lemmas:

$$\frac{\Psi;\, \Phi \vdash t : t' \qquad \Psi;\, \Phi' \vdash \sigma : \Phi}{\Psi;\, \Phi' \vdash t \cdot \sigma : t' \cdot \sigma} \qquad \frac{\Psi;\, \Phi' \vdash \sigma : \Phi \qquad \Psi;\, \Phi'' \vdash \sigma' : \Phi'}{\Psi;\, \Phi'' \vdash \sigma \cdot \sigma' : \Phi} \qquad \frac{\Psi \vdash T : K \qquad \Psi' \vdash \sigma_\Psi : \Psi}{\Psi' \vdash T \cdot \sigma_\Psi : K \cdot \sigma_\Psi}$$

(c) Substitutions over logical variables and extension variables

---

Syntax:

$$\Gamma ::= \bullet \mid \Gamma,\, x : \tau \mid \Gamma,\, x :_s \tau \mid \Gamma,\, \alpha : k \qquad e ::= \cdots \mid \mathsf{letstatic}\ x\ =\ e\ \mathsf{in}\ e'$$

Limit ctx:

$$\begin{array}{lcl} \bullet|_{\mathsf{static}} & = & \bullet \\ (\Gamma,\, x :_s t)|_{\mathsf{static}} & = & \Gamma|_{\mathsf{static}},\, x : t \\ (\Gamma,\, x : t)|_{\mathsf{static}} & = & \Gamma|_{\mathsf{static}} \\ (\Gamma,\, \alpha : k)|_{\mathsf{static}} & = & \Gamma|_{\mathsf{static}} \end{array}$$

$\Psi;\, \Sigma;\, \Gamma \vdash e : \tau$ (part)

$$\frac{\bullet;\, \Sigma;\, \Gamma|_{\mathsf{static}} \vdash e : \tau \qquad \Psi;\, \Sigma;\, \Gamma, x :_s \tau \vdash e' : \tau}{\Psi;\, \Sigma;\, \Gamma \vdash \mathsf{letstatic}\ x\ =\ e\ \mathsf{in}\ e' : \tau} \qquad \frac{x :_s \tau \in \Gamma}{\Psi;\, \Sigma;\, \Gamma \vdash x : \tau}$$

Evaluation:

$$v ::= \Lambda(K).e_d \mid \mathsf{pack}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ v \mid () \mid \lambda x : \tau.e_d \mid (v,\, v') \mid \mathsf{inj}_i\ v \mid \mathsf{fold}\ v \mid l \mid \Lambda\alpha : k.e_d$$

$$\mathcal{S} ::= \mathsf{letstatic}\ x\ =\ \bullet\ \mathsf{in}\ e' \mid \mathsf{letstatic}\ x\ =\ \mathcal{S}\ \mathsf{in}\ e' \mid \Lambda(K).\mathcal{S} \mid \lambda x : \tau.\mathcal{S} \mid \mathsf{unpack}\ e_d\ (.)x.(\mathcal{S}) \mid \mathsf{case}(e_d, x.\mathcal{S}, x.e_2)$$
$$\mid \mathsf{case}(e_d, x.e_d, x.\mathcal{S}) \mid \Lambda\alpha : k.\mathcal{S} \mid \mathsf{fix}\ x : \tau.\mathcal{S} \mid \mathsf{unify}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ (\Psi.T' \mapsto \mathcal{S}) \mid \mathcal{E}_s[\mathcal{S}]$$

$$\mathcal{E}_s ::= \mathcal{E}_s\ T \mid \mathsf{pack}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ \mathcal{E}_s \mid \mathsf{unpack}\ \mathcal{E}_s\ (.)x.(e') \mid \mathcal{E}_s\ e' \mid e_d\ \mathcal{E}_s \mid (\mathcal{E}_s,\, e) \mid (e_d,\, \mathcal{E}_s) \mid \mathsf{proj}_i\ \mathcal{E}_s \mid \mathsf{inj}_i\ \mathcal{E}_s$$
$$\mid \mathsf{case}(\mathcal{E}_s, x.e_1, x.e_2) \mid \mathsf{fold}\ \mathcal{E}_s \mid \mathsf{unfold}\ \mathcal{E}_s \mid \mathsf{ref}\ \mathcal{E}_s \mid \mathcal{E}_s := e' \mid e_d := \mathcal{E}_s \mid !\mathcal{E}_s \mid \mathcal{E}_s\ \tau$$

$e_d ::=$ all of e except $\mathsf{letstatic}\ x\ =\ e\ \mathsf{in}\ e'$ $\qquad$ $\mathcal{E} ::=$ exactly as $\mathcal{E}_s$ with $\mathcal{E}_s \to \mathcal{E}$ and $e \to e_d$

Stage 1 op.sem.:

$$\frac{(\,\mu\,,\, e_d\,) \longrightarrow (\,\mu'\,,\, e'_d\,)}{(\,\mu\,,\, \mathcal{S}[e_d]\,) \longrightarrow_s (\,\mu'\,,\, \mathcal{S}[e'_d]\,)} \qquad (\,\mu\,,\, \mathcal{S}[\mathsf{letstatic}\ x\ =\ v\ \mathsf{in}\ e]\,) \longrightarrow_s (\,\mu\,,\, \mathcal{S}[e[v/x]]\,)$$

$$(\,\mu\,,\, \mathsf{letstatic}\ x\ =\ v\ \mathsf{in}\ e\,) \longrightarrow_s (\,\mu\,,\, e[v/x]\,)$$

(d) Computational language: staging support

Figure 11. Main definitions in metatheory

We have proved that this is possible under certain restrictions: the types of the metavariables in the current context need to depend on the same free variables context $\Phi_{\max}$, or prefixes of that context. Also the substitutions they are used with need to be prefixes of the identity substitution for $\Phi_{\max}$. Such terms are characterized as collapsible. We have proved that collapsible terms can be replaced using terms that do not make use of metavariables; more details can be found in Sec. 6 and in the accompanying technical report [Stampoulis and Shao 2012].

This restriction corresponds very well to the treatment of variable contexts in the Delphin language. This language assumes an ambient context of logical variables, instead of full, contextual modal terms. Constructs to extend this context and substitute a specific variable exist. If this last feature is not used, the ambient context grows monotonically and the mentioned restriction holds trivially. In our tests, this restriction has not turned out to be limiting.

6. Metatheory

We have completed an extensive reworking of the metatheory of VeriML, in order to incorporate the features that we have presented in this paper. Our new metatheory includes a number of technical advances compared to our earlier work [Stampoulis and Shao 2010]. We will present a technical overview of our metatheory in this section; full details can be found in our technical report [Stampoulis and Shao 2012].

Variable representation technique. Though our metatheory is done on paper, we have found that using a concrete variable representation technique elucidates some aspects of how different kinds of substitutions work in our language, compared to having normal named variables. For example, instantiating a context variable with a concrete context triggers a set of potentially complicated α-renamings, which a concrete representation makes explicit. We use a hybrid technique representing bound variables as deBruijn indices, and free variables as deBruijn levels. Our technique is a small departure from the named approach, requiring fewer extra annotations and lemmas than normal deBruijn indices. Also it identifies terms not only up to α-equivalence, but also up to extension of the context with new variables; this is why it is also used within the VeriML implementation.The two fundamental operations of this technique are freshening and binding, which are shown in Fig. 11a.

Extension variables. We extend the logic with support for metavariables and context variables – we refer to both these sorts of variables as extension variables. A meta-variable X_i stands for a contextual term $T = [\Phi]\,t$, which packages a term together with the context it inhabits. Context variables ϕ_i stand for a context Φ, and are used to "weaken" parametric contexts in specific positions. Both kinds of variables are needed to support manipulation of open logical terms. Details of their definition and typing are shown in Fig. 11b. We use the same hybrid approach as above for representing these variables. A somewhat subtle aspect of this extension is that we generalize the deBruijn levels $\mathbf{I}$ used to index free variables, in order to deal effectively with parametric contexts.

Substitutions. The hybrid representation technique we use for variables renders simultaneous substitutions for all variables in scope as the most natural choice. In Fig. 11c, we show some example rules of how to apply a full simultaneous substitution σ to a term t, denoted as $t \cdot \sigma$. Similarly, we define full simultaneous substitutions σ_Ψ for extension contexts; defining their application has a very natural description, because of our variable representation technique. We prove a number of substitution lemmas which have simple statements, as shown in Fig. 11c. The proofs of these lemmas comprise the main effort required in proving the type-safety of a computational language such as the one we support, as they represent the point where computation specific to logical term manipulation takes place.

Computational language. We define an ML-style computational language that supports dependent functions and dependent pairs over contextual terms T, as well as pattern matching over them. Lack of space precludes us from including details here; full details can be found in the accompanying technical report [Stampoulis and Shao 2012]. A fairly complete ML calculus is supported, with mutable references and recursive types. Type safety is proved using standard techniques; its central point is extending the logic substitution lemmas to expressions and using them to prove progress and preservation of dependent functions and dependent pairs. This proof is modular with respect to the logic and other logics can easily be supported.

Pattern matching. Our metatheory includes many extensions in the pattern matching that is supported, as well as a new approach for dealing with typing patterns. We include support for pattern matching over contexts (e.g. to pick out hypotheses from the context) and for non-linear patterns. The allowed patterns are checked through a restriction of the usual typing rules $\Psi \vdash_p T : K$.

The essential idea behind our approach to pattern matching is to identify what the relevant variables in a typing derivation are. Since contexts are ordered, "removing" non-relevant variables amounts to replacing their definitions in the context with holes, which leads us to partial contexts $\widehat{\Psi}$. The corresponding notion of partial substitutions is denoted as $\widehat{\sigma_\Psi}$. Our main theorem about pattern matching can then be stated as:

Theorem 6.1 (Decidability of pattern matching) *If $\Psi \vdash_p T : K$, $\bullet \vdash_p T' : K$ and $\mathsf{relevant}(\Psi;\, \Phi \vdash T : K) = \widehat{\Psi}$, then either there exists a unique partial substitution $\widehat{\sigma_\Psi}$ such that $\bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi}$ and $T \cdot \widehat{\sigma_\Psi} = T'$, or no such substitution exists.*

Staging. Our development in this paper critically depends on the $\mathsf{letstatic}$ construct we presented earlier. It can be seen as a dual of the traditional box construct of Davies and Pfenning [1996]. Details of its typing and semantics are shown in Fig. 11d. We define a notion of "static evaluation contexts" $\mathcal{S}$, which enclose a hole of the form $\mathsf{letstatic}\ x = \bullet\ \mathsf{in}\ e$. They include normal evaluation contexts, as well as evaluation contexts under binding structures. We evaluate expressions e that include staging constructs using the $\longrightarrow_s$ relation; internally, this uses the normal evaluation rules, that are used in the second stage as well, for evaluating expressions which do not include other staging constructs. If stage-one evaluation is successful, we are left with a residual dynamic configuration $(\mu',\ e_d)$ which is then evaluated normally. We prove type-safety for stage-one evaluation; its statement follows.

Theorem 6.2 (Stage-one Type Safety) *If $\bullet; \Sigma; \bullet \vdash e : \tau$ then: either e is a dynamic expression e_d; or, for every store μ such that $\vdash \mu : \Sigma$, we have: either $\mu, e \longrightarrow_s \mathsf{error}$, or, there exists an e', a new store typing $\Sigma' \supseteq \Sigma$ and a new store μ' such that: $(\mu, e) \longrightarrow (\mu', e')$; $\vdash \mu' : \Sigma'$; and $\bullet; \Sigma'; \bullet \vdash e' : \tau$.*

Collapsing extension variables. Last, we have proved the fact that under the conditions described in Sec. 5, it is possible to collapse a term t into a term t' which is typed under the empty extension variables context; a substitution σ with which we can regain the original term t exists. This suggests that whenever a proof object t for a specific proposition is required, an equivalent proof object that does not mention uninstantiated extension variables exists. Therefore, we can write an equivalent proof script producing the collapsed proof object instead, and evaluate that script statically. The statement of this theorem is the following:

Theorem 6.3 *If $\Psi \vdash [\Phi]\,t : [\Phi]\,t_T$ and $\mathsf{collapsible}(\Psi \vdash [\Phi]\,t : [\Phi]\,t_T)$, then there exist Φ', t', t'_T and σ such that $\bullet \vdash \Phi'$ wf, $\bullet \vdash [\Phi']\,t' : [\Phi']\,t'_T$, $\Psi;\, \Phi \vdash \sigma : \Phi'$, $t' \cdot \sigma = t$ and $t'_T \cdot \sigma = t_T$.*

The main idea behind the proof is to maintain a number of substitutions and their inverses: one to go from a general Ψ extension context into an "equivalent" Ψ' context, which includes only definitions of the form $[\Phi]\,t$, for a constant Φ context that uses no extension variables. Then, another substitution and its inverse are maintained to go from that extension variables context into the empty one; this is simpler, since terms typed under Ψ' are already essentially free of metavariables. The computational content within the proof amounts to a procedure for transforming proof scripts inside tactics into static proof scripts.

7. Implementation

We have completed a prototype implementation of the VeriML language, as described in this paper, that supports all of our claims. We have built on our existing prototype [Stampoulis and Shao 2010] and have added an extensive set of new features and improvements. The prototype is written in OCaml and is about 6k lines of code. Using the prototype we have implemented a number of examples, that are about 1.5k lines of code. Readers are encouraged to download and try the prototype from `http://flint.cs.yale.edu/publications/supc.html`.

New features. We have implemented the new features we have described so far: context matching, non-linear patterns, proof-erasure semantics, staging, and inferencing for logical and computational terms. Proof-erasure semantics are utilized only if requested by a per-function flag, enabling us to selectively "trust" tactics. The staging construct we support is more akin to the $\langle\cdot\rangle_{\text{static}}$ form described as syntactic sugar in Sec. 5, and it is able to infer the collapsing substitutions that are needed, following the approach used in our metatheory.

Changes. We have also changed quite a number of things in the prototype and improved many of its aspects. A central change, mediated by our new treatment of the conversion rule, was to modify the used logic in order to use the explicit equality approach; the existing prototype used the λHOL_c logic. We also switched the variable representation to the hybrid deBruijn levels-deBruijn indices technique we described, which enabled us to implement subtyping based on context subsumption. Also, we have adapted the typing rules of the pattern matching construct in order to support refining the environment based on the current branch.

Examples implemented. We have implemented a number of examples to support our claims. First, we have written the type-safe conversion check routine for $\beta\mathbb{N}$, and extended it to support congruence closure based on equalities in the context. Proofs of this latter tactic are constructed automatically through static proof scripts, using a naive rewriter that is non-terminating in the general case. We have also completed proofs for theorems of arithmetic for the properties of addition and multiplication, and used them to write an arithmetic simplification tactic. All of the theorems are proved by making essential use of existing conversion rules, and are immediately added into new conversion rules, leading to a compact and clean development style. The resulting code does not need to make use of translation validation or proof by reflection, which are typically used to implement similar tactics in existing proof assistants.

Towards a practical proof assistant. In order to facilitate practical proof and program construction in VeriML, we introduced some features to support surface syntax, enabling users to omit most details about the environments of contextual terms and the substitutions used with meta-variables. This syntax follows the style of Delphin, assuming an ambient logical variable environment which is extended through a construct denoted as $\nu x : t.e$. Still, the full power of contextual modal type theory is available, which is crucial in order to change what the current ambient environment is, used, as we saw earlier, for static calls to tactics. In general the surface syntax leads to much more concise and readable code.

Last, we introduced syntax support for calls to tactics, enabling users to write proof expressions that look very similar to proof scripts in current proof assistants. We developed a rudimentary ProofGeneral mode for VeriML, that enables us to call the VeriML type-checker and interpreter for parts of source files. By adding holes to our sources, we can be informed by the type inference mechanism about their expected types. Those types correspond to what the current "proof state" is at that point. Therefore, a possible workflow for developing tactics or proofs, is writing the known parts, inserting holes in missing points to know what remains to be proved, and calling the typechecker to get the proof state information. This workflow corresponds closely to the interactive proof development support in proof assistants like Coq and Isabelle, but generalizes it to the case of tactics as well.

8. Related work

There is a large body of work that is related to the ideas we have presented here.

Techniques for robust proof development. There have been multiple proposals for making proof development inside existing proof assistants more robust. A well-known technique is *proof-by-reflection* [Boutin 1997]: writing total and certified decision procedures within the functional language contained in a logic like CIC. A recently introduced technique is *automation through canonical structures* [Gonthier et al. 2011]: the resolution mechanism for finding instances of canonical structures (a generalization of type classes) is cleverly utilized in order to program automation procedures for specific classes of propositions. We view both approaches as somewhat similar, as both are based in cleverly exploiting static "interpreters" that are available in a modern proof assistant: the partial evaluator within the conversion rule in the former case; the unification algorithm within instance discovery in the latter case.

Our approach can thus be seen as similar, but also as a generalization of these approaches, since a general-purpose programming model is supported. Therefore, users do not have to adapt to a specific programming style for writing automation code, but can rather use a familiar functional language. Proof-by-reflection could perhaps be used to support the same kind of extensions to the conversion rule; still, this would require reflecting a large part of the logic in itself, through a prohibitively complicated encoding. Both techniques are applicable to our setting as well and could be used to provide benefits to large developments within our language.

The style advocated in Chlipala [2011] (and elsewhere) suggests that proper proof engineering entails developing sophisticated automation tactics in a modular style, and extending their power by adding proved lemmas as hints. We are largely inspired by this approach, and believe that our introduction of the extensible conversion rule and static checking of tactics can significantly benefit it. We demonstrate similar ideas in layering conversion tactics.

Traditional proof assistants. There are many parallels of our work with the *LCF family of proof assistants*, like HOL4 [Slind and Norrish 2008] and HOL-Light [Harrison 1996], which have served as inspiration. First, the foundational logic that we use is similar. Also, our use of a dedicated ML-like programming language to program tactics and proof scripts is similar to the approach taken by HOL4 and HOL-Light. Last, the fact that no proof objects need to be generated is shared. Still, checking a proof script in HOL requires evaluating it fully. Using our approach, we can selectively evaluate parts of proof scripts; we focus on conversion-like tactics, but we are not limited inherrently to those. This is only possible because our proof scripts carry proof state information within their types. Similarly, proof scripts contained within LCF tactics cannot be evaluated statically, so it is impossible to establish their validity upon tactic definition. It is possible to do a transformation similar to ours manually (lifting proof scripts into auxiliary lemmas that are proved prior to the tactic), but the lack of type information means that many more details need to be provided.

The Coq proof assistant [Barras et al. 2010] is another obvious point of reference for our work. We will focus on the conversion rule that CIC, its accompanying logic, supports – the same problems with respect to proof scripts and tactics that we described in the LCF case also apply for Coq. The conversion rule, which identifies computationally equivalent propositions, coupled with the rich type universe available, opens up many possibilities for constructing small and efficiently checkable proof objects. The implementation of the conversion rule needs to be part of the trusted base of the proof assistant. Also, the fact that the conversion check is built-in to the proof assistant makes the supported equivalence rigid and non-extensible by frequently used decision procedures.

There is a large body of work that aims to extend the conversion rule to arbitrary confluent rewrite systems (e.g. Blanqui et al. [1999]) and to include decision procedures [Strub 2010]. These approaches assume some small or larger addition to the trusted base, and extend the already complex metatheory of Coq. Furthermore, the NuPRL proof assistant [Constable et al. 1986] is based on extensional type theory which includes an extensional conversion rule. This enables complex decision procedures to be part of conversion; but it results in a very large trusted base. We show how, for a subset of these type theories, the conversion check can be recovered outside the trusted base. It can be extended with arbitrarily complex new tactics, written in a familiar programming style, without any metatheoretic additions and without hurting the soundness of the logic. The question of whether these type theories can be supported in full remains as future work, but as far as we know, there is no inherrent limitation to our approach.

Dependently-typed programming. The large body of work on dependently-typed languages has close parallels to our work. Out of the multitude of proposals, we consider the Russell framework [Sozeau 2006] as the current state-of-the-art, because of its high expressivity and automation in discharging proof obligations. In our setting, we can view dependently-typed programming as a specific case of tactics producing complex data types that include proof objects. Static proof scripts can be leveraged to support expressivity similar to the Russell framework. Furthermore, our approach opens up a new intriguing possibility: dependently-typed programs whose obligations are discharged statically and automatically, through code written within the same language.

Last, we have been largely inspired by the work on languages like Beluga [Pientka and Dunfield 2008] and Delphin [Poswolsky and Schürmann 2008], and build on our previous work on VeriML [Stampoulis and Shao 2010]. We investigate how to leverage type-safe tactics, as well as a number of new constructs we introduce, so as to offer an extensible notion of proof checking. Also, we address the issue of statically checking the proof scripts contained within tactics written in VeriML. As far as we know, our development is the first time languages such as these have been demonstrated to provide a workflow similar to interactive proof assistants.

Acknowledgments

We thank anonymous referees for their suggestions and comments on an earlier version of this paper. This research is based on work supported in part by DARPA CRASH grant FA8750-10-2-0254 and NSF grants CCF-0811665, CNS-0910670, and CNS 1065451. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

H.P. Barendregt and H. Geuvers. Proof-assistants using dependent type systems. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Sci. Pub. B.V., 1999.

B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.C. Filliâtre, E. Giménez, H. Herbelin, et al. The Coq proof assistant reference manual (version 8.3), 2010.

F. Blanqui, J.P. Jouannaud, and M. Okada. The calculus of algebraic constructions. In *Rewriting Techniques and Applications*, pages 671–671. Springer, 1999.

F. Blanqui, J.P. Jouannaud, and P.Y. Strub. A calculus of congruent constructions. *Unpublished draft*, 2005.

S. Boutin. Using reflection to build efficient and certified decision procedures. *Lecture Notes in Computer Science*, 1281:515–529, 1997.

A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 2011 ACM SIGPLAN conference on Programming Language Design and Implementation*. ACM, 2011.

R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ, 1986.

R. Davies and F. Pfenning. A modal analysis of staged computation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 258–270. ACM, 1996.

G. Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55 (11):1382–1393, 2008.

G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. In *Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming*, pages 163–175. ACM, 2011.

J. Harrison. HOL Light: A tutorial introduction. *Lecture Notes in Computer Science*, pages 265–269, 1996.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 207–220. ACM, 2009.

X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

T. Nipkow, L.C. Paulson, and M. Wenzel. Isabelle/HOL : A Proof Assistant for Higher-Order Logic, volume 2283 of LNCS, 2002.

B. Pientka and J. Dunfield. Programming with proofs and explicit contexts. In *Proceedings of the 10th international ACM SIGPLAN conference on Principles and Practice of Declarative Programming*, pages 163–173. ACM New York, NY, USA, 2008.

A. Poswolsky and C. Schürmann. Practical programming with higher-order encodings and dependent types. *Lecture Notes in Computer Science*, 4960:93, 2008.

V. Siles and H. Herbelin. Equality is typable in semi-full pure type systems. In *2010 25th Annual IEEE Symposium on Logic in Computer Science*, pages 21–30. IEEE, 2010.

K. Slind and M. Norrish. A brief overview of HOL4. *Theorem Proving in Higher Order Logics*, pages 28–32, 2008.

M. Sozeau. Subset coercions in coq. In *Proceedings of the 2006 International Conference on Types for Proofs and Programs*, pages 237–252. Springer-Verlag, 2006.

A. Stampoulis and Z. Shao. VeriML: Typed computation of logical terms inside a language with effects. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 333–344. ACM, 2010.

A. Stampoulis and Z. Shao. Static and user-extensible proof checking (extended version). Available in the ACM Digital Library, 2012.

P.Y. Strub. Coq modulo theory. In *Proceedings of the 24th International Conference on Computer Science Logic*, pages 529–543. Springer-Verlag, 2010.