Dissertation Summary

"VeriML: A dependently-typed, user-extensible and language-centric approach to proof assistants"

Antonios Michael Stampoulis

1 Introduction

The mechanization of various formal logics and their extension into proof assistants such as Coq [1] and Isabelle [2] has enabled the success of a number of large-scale formal proof development projects [e.g. 3, 4]. Yet formal proof development still requires significant manual effort, due to various shortcomings of existing proof assistants. Oft-cited issues include the difficulty of implementing and maintaining new tactics [5]; and the fact that small-scale automation – the component of a proof assistant that handles trivial proof steps transparently – lacks extensibility [6]. This dissertation revisits the foundations of proof assistants, attempting to address the fundamental reasons behind such issues.

Specifically, we propose an alternative architecture for future proof assistants based on a new programming language design called VeriML. VeriML constitutes a single meta-language where we write all programs that perform computations on logical terms (e.g., tactics, large- and small-scale automation procedures). This runs counter to current state-of-the-art proof assistants which require the use of multiple languages with different tradeoffs. This language-based approach is reminiscent of the LCF tradition to proof assistants, yet VeriML departs in one key way: the type system of the meta-language retains all typing information about logical terms, as determined by the (object) logic type system. Type safety then guarantees, for example, that a program which claims to produce a proof of a specific proposition, indeed produces a valid proof of that same proposition upon successful evaluation. This guarantee is true even if the program contains side-effectful code.

The rich type information available in VeriML has far-reaching consequences. First, it increases the modularity and maintainability of tactics and automation procedures, as the relation between input and output logical terms is evident at the type level. Second, the typing information of incomplete programs – e.g. the available hypotheses, the proofs that remain to be provided – can be presented to the user to help them proceed; this is a generalization of the interactive use of current proof assistants for proof development, to the domain of tactic and automation procedure development as well. Third, it allows better reuse of proof scripts, because the initial proof state they depend on, as well as its evolution through the script, is available at the level of types. Last, by staging calls to existing automation procedures so as to happen at compile time, VeriML allows us to perform further static checks that go beyond the VeriML type system. This significantly simplifies the development of further automation procedures. We have shown that through careful layering of automation procedures, the required manual proof effort is greatly reduced.

Perhaps the most important consequence of the VeriML design from a formal standpoint, is that it enables the development of an *extensible conversion rule*, which allows safe, user-defined extensions. The suggested approach also elucidates the differences between the conversion rule in existing systems, which can potentially help in the long-standing debate about equality in Martin-Löf type theories and related systems. The main insight behind our approach is that the conversion rule can be viewed as a way to avoid generating proof witnesses for definitional

equality, in the interest of implementation efficiency. This is done by hard-coding a trusted decision procedure for equality in the core implementation of the mechanized logic. Yet the logic-related aspect – namely, what definitional equality actually is – needs to be separated from the implementation-related aspect – what proof strategy to use to decide it, while avoiding large witnesses that occur ubiquitously. The type system of VeriML allows us to develop similar trusted tactics both for definitional equality and for predicates that go beyond it. Type safety then guarantees that if such tactics succeed, the associated proof witnesses exist, even if they are not actually generated at runtime. We show that through this approach we can develop VeriML tactics to recover the benefits of the conversion rule, starting from a logic that does not include it, as well as extend it with equational and arithmetical reasoning, all the while without requiring extensions to the metatheory or implementation of the base logic.

The dissertation presents all aspects of the VeriML language, including the formal design, the metatheory, an extensive prototype implementation of the language, and a number of proofs and tactics developed using the language. Technically, the VeriML language is an extension of a core ML-style calculus with dependently-typed constructs for introducing and manipulating logical terms through pattern matching. The main challenge in this extension is how to allow such constructs to work with open logical terms inhabiting different variable contexts. We address this challenge through an adaptation of contextual modal type theory; our development presents a number of technical advances in the treatment of variables and the presentation of pattern matching compared to existing approaches. The development maintains orthogonality between the logic-related constructs and the existing constructs of the ML core calculus, so that the system can be extended in both directions.

2 The VeriML language design

Logical layer. A *mechanized logic* is an implementation of a specific formal logic as a computer system. This implementation at the very least consists of a way to represent the terms and proofs (or derivations) of the logic as computer data; and also of a procedure that checks whether a claimed derivation is indeed valid, according to the rules of the logic. We refer to the machine representation of a specific logical derivation as the *proof object*; the computer code that decides the validity of proof objects is called the *proof checker*. We use this latter term in order to signify the trusted core of the implementation of the logic: bugs in this part of the implementation might lead to invalid proofs being admitted, destroying the logical soundness of the overall system.

In the dissertation, we use the logic λHOL as our formal logic; our VeriML implementation includes a mechanized version of this logic as the core logical layer. λHOL is a simple type-theoretic higher-order logic based on System Fω. It was first introduced in [7] and can be seen as a shared logical core between CIC [8] and the HOL family of logics as used in proof assistants such as HOL4 [9], HOL-Light [10] and Isabelle/HOL [11]. It is a constructive logic yet admits classical axioms.

The logic is composed of the following classes:

- The **objects of the domain of discourse**, denoted d: these are the objects that the logic reasons about, such as natural numbers, lists and functions between such objects.
- The **propositions**, denoted as P: these are the statements of the logic. They include implication $P_1 \rightarrow P_2$, quantification over domain objects $\forall x : \mathcal{K}.P$ and propositional equality $d_1 = d_2$. Other connectives and predicates can be inductively defined. As this is a higher-order logic, propositions themselves are objects of the domain of discourse d, having a special *Prop* type. We use P instead of d when it is clear from the context that a domain object is a proposition.
- The **kinds**, denoted $\mathcal{K}$: these are the classifiers of the objects of the domain of discourse, including the *Prop* kind for propositions and the $A \rightarrow B$ kind for total functions from type A to B.
- The **sort** *Type*, used to classify kinds.

– The **proofs**, denoted π, which correspond to a linear representation of valid derivations of propositions.

The rules of the logic are formulated as a type system and are given in Chapter 3 of the dissertation. For example, the judgement $\Phi \vdash \pi : P$ specifies the rules for deciding that the proof π is a valid proof for the proposition P in the variable context Φ. The mechanization of this judgement is the proof checker. Other judgements include kinding of domain objects, $\Phi \vdash d : \mathcal{K}$ which includes valid propositions $\Phi \vdash P : \mathit{Prop}$ as a special case; and valid kinds $\Phi \vdash \mathcal{K} : \mathit{Type}$.

In the rest, we refer to terms of the above layers collectively as *logical terms*, denoted as t; and use an equivalent collapsed formulation of the logical judgements, denoted as $\Phi \vdash t : t'$.

Proof assistant. A mechanized logic automates the process of validating formal proofs, but does not help with actually developing such proofs. This is what the systems referred to as *proof assistants* are used for. Examples of proof assistants include Coq [1], Isabelle [2], HOL4 [9], NuPRL [12], and ACL2 [13]. At its core, each proof assistant contains a specific mechanized logic together with an interface for adding definitions and setting goals to be proved. Most importantly, the proof assistant contains library of *proof-producing functions* – also called tactics, decision procedures and automated theorem provers – which are used so as to arrive to proof objects for each goal. Most proof assistants also offer one or more meta-languages used for developing new proof-producing functions, as well as composing such functions together into *proof scripts*. Proof scripts can be developed interactively by asking the proof assistant for the current *proof state* – the existing hypotheses and the goals that are yet to be proved.

When a proof script is complete, we can run it in order to produce a proof object, which is checked for validity using the proof checker. This is the mode of operation in the Coq proof assistant and in Isabelle/HOL, but other architectures are possible. In proof assistants following the original LCF tradition [14] such as HOL4 and HOL-Light, the type safety of the meta-language where proof functions are written guarantees that valid proof objects exist, even if they are never actually produced. In other assistants, such as ACL2, proof objects are not guaranteed to exist and the proof functions are trusted for validity; the trusted base of the system thus contains not only the mechanized logic but such functions as well.

Basic design of VeriML. VeriML follows a language-centric approach to proof assistants: it is a single meta-language that containts constructs for performing computation with logical terms; proof-producing functions and proof scripts are expressions in this language with specific types. This is reminiscent of the LCF tradition to proof assistants which uses ML as the meta-language. In the rest, we denote expressions of the meta-language as e and its types as τ; we also refer to the meta-language as the *computational language*.

The key departure of VeriML compared to LCF is that *the type of logical terms at the meta-level retains the typing information from the logical level*. Thus, proof scripts and tactics coded in VeriML carry type information about the logical terms they require and the logical terms they produce. For example, an automated theorem proving procedure would have the following type in VeriML:

$$\mathsf{auto} : (P : \mathit{Prop}) \rightarrow \mathsf{option}\ P$$

That is, given a proposition P, the procedure will produce a proof for that proposition or return failure – denoted through the normal ML option type. Type safety of the computational layer then ensures that upon successful evaluation, not only will the proof returned by auto be a valid proof (as in traditional LCF), but it will prove the input proposition P specifically.

Alternatively, we say that the constructs for computing with logical terms at the meta-level are dependently typed: their static type τ contains information about their runtime value – namely, the logical type t as determined through the logical type system. We can thus view VeriML as a dependently-typed version of the LCF tradition of proof assistants. Formally, the typing judgement of the logical layer $\Phi \vdash_{\lambda\mathrm{HOL}} t : t'$ is used as part of the typing judgements of the computational layer. The proof checker for the logical layer, instead of operating at runtime, is used by the type checker for the computational layer statically.

The increased type information enhances the maintainability and modularity of tactics and proof scripts, and also enables techniques that facilitate tactic development as we show in Chapters 7 and 9 of the dissertation. Development of user-defined domain-specific tactics is often argued for as a way to minimize the proof effort required in large developments [e.g. 15]; the lack of good language support for doing this is one of the main motivations behind the VeriML design.

Open terms. The main constructs of the computational layer that involve logical terms are functions and tuples over logical terms (corresponding to the dependent product and sum types); and a pattern matching construct for looking into the structure of logical terms. Proof-producing functions inspect their arguments through pattern matching and produce the appropriate logical terms – simplified versions of input propositions, proofs, further goals to be proved etc. It is important for such constructs to support *open terms*: logical terms that depend on different variable contexts. Such examples are proofs under a set of hypotheses and quantified variables and the body of a quantified proposition. Supporting typing of open logical terms and pattern matching against them is the main technical challenge of the VeriML design. Our solution is based on contextual modal type theory [16] and is presented in Chapters 4 and 5, forming the main technical core of the dissertation.

The idea of contextual modal type theory is to explicitly capture the dependence on variable contexts in the type of terms; to that end, we introduce a notion of *contextual terms*, which package logical terms (or types) together with the context they depend on and are denoted as $[\Phi]\, t$. In this way, we can assign meaningful types to procedures involving terms in different contexts, such as the Cut tactic:

$$\mathsf{Cut} : (\Phi) \to (\mathit{Goal} : [\Phi]\, \mathit{Prop}) \to (\mathit{Hyp} : [\Phi]\, \mathit{Prop}) \to ([\Phi]\, \mathit{Hyp}) \to ([\Phi, H : \mathit{Hyp}]\, \mathit{Goal}) \to ([\Phi]\, \mathit{Goal})$$

The type of this tactic can be read as: assume a variable context Φ, a goal and a hypothesis that make sense in that context; then, if we provide the tactic with a proof of the hypothesis in the original context Φ, and a proof of the goal assuming the hypothesis – a proof of the goal in the context Φ extended with a new variable H for the hypothesis –, then the tactic will return a proof of the goal in the original Φ context.

Proof scripts, proof state and interactivity. When using tactics such as Cut and Auto in an expression, the available typing information can be used to infer some of the arguments; special syntax can be provided to facilitate the use of such tactics. In this way, users can write proof scripts that are similar to the ones offered in traditional proof assistants. The types that are determined for the remaining arguments might be helpful to the user for filling them in. For example, a user might write the following partial proof script expression:

$$\begin{array}{ll} \mathsf{let}\ \mathit{proof1} : (P \to Q \to R) \to (P \to Q) \to P \to R = & \mathsf{Intro}\ H1, H2, H3\ \mathsf{in} \\ & \mathsf{Cut}\ H4 : (Q \to R)\ \mathsf{by\ Auto\ in} \\ & \mathsf{Cut}\ H5 : Q\ \mathsf{by\ Auto\ in} \\ & ? \end{array}$$

where we have assumed that Auto will only use hypotheses to solve a goal G if they are of the form G or $H \to G$ for a single premise H, and that ? represents a missing expression. We can query the VeriML type checker for the expected type of the missing expression:

$$[H1 : P \to Q \to R, H2 : P \to Q, H3 : P, H4 : Q \to R, H5 : Q]\, R$$

From this, the user can tell that Auto should be able to solve the goal and can use it to fill in the missing expression.

Thus in VeriML, the notion of *proof state* (which represents the current hypotheses and missing goals) is *explicitly present at the level of types of the meta-language*. In traditional proof assistants, this information is hidden as an imperative data structure that is manipulated by tactics; in order to explore its evolution throughout a proof script, the user has to execute it step-by-step. This enhances the reusability of proof scripts in VeriML, as there is more information available about the cases in which they apply. Furthermore, the support of interactive development of

proof scripts is recovered as interactive querying of the type checker. This interactivity support is not limited to writing proof scripts, but can be used when writing tactics as well, generalizing the support available in traditional proof assistants

Pattern matching. Proof-producing functions are written by making use of pattern matching to inspect their arguments. VeriML allows pattern matching both over open terms and over variable contexts as well; and also various practically useful features such as non-linear patterns, simultaneous matching and context refinement, as described in Chapter 5 and Section 6.2 of the dissertation. It is thus quite similar to the support available in languages such as LTac [17] (the language available in Coq for writing user-defined tactics).

The pattern matching construct is dependently-typed, meaning that the specific pattern used in each branch informs the typing of its body. For example, consider the following version of the Auto tactic:

$$
\begin{array}{lcl}
\mathsf{Auto} & : & (\Phi) \to (P : \mathit{Prop}) \to \mathsf{option}\,([\Phi]\,P) \\
\mathsf{Auto}\;\Phi\;P & = & \mathsf{match}\;P\;\mathsf{with} \\
 & & \quad Q \wedge R \;\mapsto\; \mathsf{do}\;\; X \leftarrow \mathsf{Auto}\;\Phi\;\langle\, Q \,\rangle\;; \\
 & & \qquad\qquad\quad Y \leftarrow \mathsf{Auto}\;\Phi\;\langle\, R \,\rangle\;; \\
 & & \qquad\qquad\quad \mathsf{return}\;?_1 \\
 & & \mid Q \vee R \;\mapsto\; (\mathsf{do}\;\; X \leftarrow \mathsf{Auto}\;\Phi\;\langle\, Q \,\rangle\;; \\
 & & \qquad\qquad\quad\;\; \mathsf{return}\;?_2)\;\|\| \\
 & & \qquad\qquad\quad (\mathsf{do}\;\; Y \leftarrow \mathsf{Auto}\;\Phi\;\langle\, R \,\rangle\;; \\
 & & \qquad\qquad\quad\;\; \mathsf{return}\;?_3) \\
 & & \mid Q \to R \;\mapsto\; \mathsf{do}\;\; X \leftarrow \mathsf{Auto}\;(\Phi,\, Q)\;\langle\, R \,\rangle\;; \\
 & & \qquad\qquad\quad \mathsf{return}\;?_4
\end{array}
$$

In each branch, the extra knowledge about the structure of P will be reflected in the expected type for the proof that needs to be returned – for example, the expected type for the missing proof $?_1$ will be $[\Phi]\,Q \wedge R$. In this way errors in tactics are caught statically.

ML core. Other than the logic-related constructs we have mentioned so far, the computational level of VeriML also includes an ML-style core calculus, consisting of higher-order functions, algebraic datatypes, general recursion and mutable state. VeriML maintains a strict separation between the two levels: the logical level terms can be manipulated by the computational level, stored in computational data structures and references; but the computational level terms cannot directly be used inside logical terms and thus the core logic remains unchanged.

Still, the presence of the ML core calculus means that efficient data structures involving logical terms can be created and also that tactics can make use of side-effectful operations in a safe manner. For example, we have implemented an efficient imperative union-find data structure which is used to store information about equalities between terms; this is then used to develop a decision procedure for equality with uninterpreted functions, as presented in Section 9.4 of the dissertation. Of course, the ability to use side-effects means that VeriML expressions are not bound to terminate; therefore the types of tactics and proof scripts are not interpreted to mean that the claimed proofs are bound to exist, but rather that upon successful evaluation, the claimed proofs will be produced.

Comparison with traditional meta-languages. VeriML combines type safety, a full, side-effectful programming model and and convenient pattern matching constructs over logical terms. This combination is a unique characteristic of VeriML compared to the meta-languages of current proof assistants; this is the main contribution of the dissertation. The meta-languages and techniques traditionally used for writing tactics offer instead one of these three features in isolation:

– The LTac language used in Coq has convenient pattern-matching support but lacks a side-effectful programming model, support for data structures such as arrays and hashtables and is completely untyped.

– Using ML to program tactics as in the original LCF approach allows the use of a full programming model so efficient tactics can be programmed. Yet logical terms are not first class but rather are represented using the general-purpose ML constructs. This means that the pattern-matching constructs available, while adequate, are not especially well-tailored to tactic programming as in LTac or VeriML; and also, that the type system does not capture any information about the logical terms. The guarantees that type safety provides for tactics and proof scripts are thus severely limited.

– Last, proof assistants such as Coq make use of the proof-by-reflection technique, which consists of writing tactic-like procedures directly in the logical level. Though this gives increased type information and static guarantees – namely, total correctness, rather than just partial correctness as in VeriML – it precludes the use of side-effects. Also, the technique requires tedious encodings of logical terms as inductive definitions, which further limit its applicability and make it especially difficult to use when dealing with open terms.

The fact that no single language combines these features means that various developments in current proof assistants use a combination involving all three languages and techniques in order to achieve the desired result. Interoperation between these approaches is problematic, leading to developments that are hard to extend and maintain. Combining these features in a single language as in VeriML thus greatly simplifies tactic programming. We demonstrate this in Chapter 9, through the development of a congruence closure procedure and an arithmetic simplification procedure.

Extensible static checking. VeriML includes a simple staging construct, that allows users to evaluate closed VeriML expressions statically. Staging, coupled together with a code transformation called *collapsing* which is based on the rich type information available in VeriML, enables *extensible static checking*: the ability to statically validate properties beyond what the base VeriML type system allows, through code written in VeriML itself. For example, by developing a set of tactics for deciding equality between two logical terms, we can statically check equality even though the VeriML type system does not have explicit support for that.

Consider the following tactic:

$$\mathsf{Exact} : (P, Q : \mathit{Prop}) \rightarrow (P = Q) \rightarrow (P) \rightarrow (Q)$$

Using this tactic requires a proof that the two propositions P and Q involved are equal. The static checking technique that VeriML introduces, allows us to perform a staged call to a globally-defined equality tactic Equal to discover this proof. Such staged calls can happen both when writing proof scripts and when writing tactics. Thus, from a user perspective the static checking available in VeriML is effectively extended through the Equal tactic. Its functionality can be further extended to apply to user-defined domains as well. The extensible conversion rule we will cover shortly stems from this idea.

The extensibility inherrent in this technique makes it a very good fit for handling trivial details of proofs – propositional equality is only one such possibility; other predicates and arbitrary propositions can be handled through this technique as well. We show that this technique significantly reduces the amount of trivial proof steps required, by creating appropriate *layers* of tactics that handle such trivial details. Each layer helps the development of the higher layers, in a manner reminiscent of how the build-up of mathematical sophistication allows skipping increasingly more details in informal proofs. The aforementioned examples of congruence closure and arithmetic simplification are indeed written in this layered style.

Proof erasure. We have proved that type safety of VeriML implies another interesting metatheoretic property: the fact that proofs can be erased from VeriML expressions prior to evaluation, yet *valid proof objects are guaranteed to*

exist. This fact follows by disallowing pattern matching against proof terms; therefore runtime evaluation cannot depend on the specific structure of proofs. We refer to this feature of VeriML as *proof erasure*. Rather than using proof erasure for evaluating all VeriML expressions, we allow users to selectively interpret certain tactics under proof erasure as they see fit. This allows users a wide spectrum of choices with regards to what the trusted base of the system is: disallowing uses of proof erasure means that the trusted base is only the basic proof checker of the λHOL logic; using full proof erasure means that they trust the VeriML implementation in its entirety. The intermediate points consist of trusting the proof-erased versions of certain tactics. We will later show how this is of crucial importance in implementing the extensible conversion rule.

3 The extensible conversion rule

The treatment of equality is a contentious issue in Martin-Löf type theory, with a long-standing debate between its two main versions, namely intensional and extensional. In practice, most proof assistants based on type theory implement a version of the intensional approach with slight variations, whereas proof assistants in the NuPRL family implement extensional type theory. In the dissertation, we revisit this debate in light of the VeriML design, decoupling the issue of logical expressivity from implementation issues; as a result, we offer a novel treatment of the conversion rule inside VeriML that can be used to unify and extend these approaches. We summarize our approach in this section.

In order to present the two basic approaches to equality, we first need to distinguish between two different notions of equality: propositional vs. definitional equality. Definitional equality relates terms that are indistinguishable from each other in the logic. Proving that two definitionally equal terms are equal is trivial through reflexivity, since they are viewed as exactly the same terms. We will use $d_1 \equiv d_2$ to represent definitional logic. Propositional equality is an actual predicate of the logic. Terms are proved to be propositionally equal using the standard axioms and rules of the logic. In λHOL we would represent propositional equality through propositions of the form $d_1 = d_2$. With this distinction in place, we will give a brief summary of the different approaches below.

Intensional approach. In intensional type theories such as CIC as used in Coq, two terms are deemed definitionally equal if they are identical up to partial evaluation of the total functions they contain (referred to as $\beta\iota$-equality). Because of the totality restriction on functions, definitional equality is decidable. In this case, definitional equality is usually presented through the *conversion rule*:

$$\frac{\Phi \vdash t : t' \qquad t' \equiv t''}{\Phi \vdash t : t''}$$

An alternate presentation is also possible, usually referred to as judgemental equality, where definitional equality is presented as a judgement $\Phi \vdash t' \equiv t'' : s$ and the conversion rule is instead written as:

$$\frac{\Phi \vdash t : t' \qquad \Phi \vdash t' \equiv t'' : s}{\Phi \vdash t : t''}$$

It has been argued [e.g. 18] that the latter presentation facilitates the construction of a model for the logic and is therefore advantageous in terms of metatheoretic reasoning.

Extensional approach. This approach, which is followed in proof assistants such as NuPRL, includes propositional equality as part of definitional equality. This can be viewed essentially as extending the the definitional equality judgement with the following rule:

$$\frac{\Phi \vdash \pi : t = t'}{\Phi \vdash t \equiv t' : s}$$

> The addition of this rule adds significant new expressivity to the type theory (e.g. functional extensionality is provable); yet it renders definitional equality and therefore typechecking undecidable, as inhabitation of propositional equality is undecidable in the general case.

Recently-proposed systems such as the Calculus of Algebraic Constructions [19] and Coq Modulo Theories [20] suggest a third approach between these two extremes. They extend the intensional approach with a richer notion of definitional equality that includes the ability to use first-order rewriting lemmas, such as arithmetic simplification rules, yet maintain decidability of type checking.

The discussion regarding these alternatives involves three main issues: first, the logical expressivity of the resulting type theory; second, whether type checking is decidable; and third, what is the size of the resulting proof objects. For example, if the intensional approach with judgemental equality is followed and full derivations witnessing this equality are stored as part of proof objects, their resulting size is prohibitively large and is thus practically impossible. Another important issue is the implications of each approach with regards to the trusted core of the system. The proof checker needs to include a way to decide definitional equality. Enlarging the notion of definitional equality therefore results in an increase in the trusted core.

Following [21] we argue that decidability has been overemphasized in this debate. For example, in several large developments, a simple implementation of the basic definitional equality (e.g. through call-by-need interpretation) is almost as bad as undecidable type checking – the conversion problems that come up cannot be effectively decided. The same problems might be easy to decide using a different strategy (e.g. through compilation to native code), yet adding such a strategy as part of the proof checker of the system results in a significantly enlarged trusted core.

Based on these concerns, we posit that the standard presentation of the conversion rule is to blame, as it conflates together some of the issues mentioned above. Rather, we can follow the judgemental equality approach and understand proof objects to be representations of full derivations according to the rules of the logic, including witnesses for judgemental equality. In this way, proof checking is always decidable since all the necessary information is available. We are thus free to make the choice of what judgemental equality contains based on the desired logical expressivity and the intended semantic model of the logic. Yet as we mentioned above, keeping witnesses for definitional equality soon becomes impractical. This is the problem that the conversion rule essentially solves, by replacing such derivations by computation.

The main insight of our approach is to view the conversion rule as a *trusted decision procedure* decideEqual*, hardcoded within the proof checker, which decides whether a witness for definitional equality exists, without producing these witnesses*. Assuming that the proof checker is implemented in ML, we can thus regard the following as an alternative presentation of the conversion rule:

$$\frac{\mathsf{decideEqual}\ t\ t' \longrightarrow_{\mathrm{ML}} \mathsf{true}}{\Phi \vdash t = t' : s}$$

Viewed in this light, the question about decidability becomes a question whether a complete and total decision procedure decideEqual exists. In extensional type theory, where judgemental equality is undecidable, no such procedure exists; yet the actual implementation of the proof checker includes a number of decision procedures that can effectively decide some subset of it. In intensional type theory, the built-in decision procedure can in principle decide the full subset, but in practice only a subset can be effectively decided. Thus the question of which subset of judgemental equality can be treated effectively through the built-in decideEqual procedure deserves similar attention to the question of decidability.

This discussion suggests that having such a fixed decision procedure as the conversion rule within the proof checker is limiting: the procedure will never be able to effectively treat all equality problems that arise in proof developments. A better approach would be to allow the conversion rule to be *extensible*, so that users (or implementors) can define new strategies for deciding equality, yet without modifying the internals of the system and enlarging the trusted core.

We show that such an *extensible conversion rule* is possible in VeriML, if the conversion rule is implemented in the *meta-language* instead of being part of the logic. Instead of having a hard-coded trusted tactic as part of the proof checker, we can write VeriML tactics to decide equality. By assigning the appropriate type to these tactics, we make sure that equality witnesses exist, even if the tactics are evaluated under proof-erasure semantics and the actual witnesses are never actually produced. VeriML allows us to assign the following type to the decideEqual tactic:

$$\mathsf{decideEqual} : (\Phi) \to (T : \mathit{Type}) \to (t, t' : T) \to \mathsf{option}\ (t = t')$$

A conversion check is successful when the following holds, using the proof-erased version of VeriML semantics, denoted as $\longrightarrow^{\text{erase}}_{\text{VeriML}}$:

$$\mathsf{decideEqual}\ \Phi\ t\ t' \longrightarrow^{\text{erase}}_{\text{VeriML}} \mathsf{Some}\ ()$$

The conversion rule is then implemented outside the logic, through the definition of a notion of *proof object expressions* – VeriML expressions that are allowed to use normal proof objects and calls to such conversion tactics.

Through this approach, users are free to write their own conversion tactics as normal VeriML code. They also are free to decide what granularity they want the produced proof witnesses to have, choosing between efficiency or trustworthiness, by selecting which conversion tactics to evaluate under proof erasure and which to evaluate under normal semantics. Thus the decision of what the conversion rule actually is rests with users instead of the designer of the logic, but logical soundness is maintained thanks to the rich types that VeriML allows.

In Section 7.1 of the dissertation, we demonstrate this approach to conversion by showing the equivalence of a version of λHOL with a traditional conversion rule and a version where explicit equality is used. The λHOL logic does not have a distinction between propositional and definitional equality as discussed above, so we cast this question in terms of propositional equality; but our approach can be generalized to type theories with such a distinction. In Section 7.2 we also show that the extensible conversion rule is applicable not only in closed proof scripts but inside tactics as well, through the use of staging. We argue about how this renders the extensible conversion rule as a user-extensible approach to small-scale automation.

Last, in the examples we have implemented and describe in Chapter 9, we progressively develop more sophisticated versions of conversion, starting from the version of λHOL without a conversion rule. We develop β-equality, rewriting based on first-order lemmas, congruence closure and arithmetic simplifications, all the while without changing the trusted core of the system or requiring extensions to the soundness proof of the underlying logic. We thus believe that our approach compares favorably to approaches such as CoqMT. Each new conversion check that we develop benefits from existing conversion rules, simplifying the involved proofs. In this way we show that VeriML can be used to develop verified tactics in a manner that is easier than developing untyped tactics in current systems, through the techniques that the increased type information enables.

4 Structure of the dissertation and main results

In this section we will present a summary of the main technical results of the dissertation. We will start by describing the design, implementation and usage examples of the computational VeriML language with more formal details than above, following Chapters 6 through 9 of the dissertation. Our technical development is relatively orthogonal to the logic that is used as the logical layer, save for certain assumptions about the theorems and operations available. We will then summarize the content of Chapters 3 through 5 which demonstrate how the λHOL logic is made to satisfy these assumptions. Through similar techniques as the ones we will present, the VeriML design could be carried out for logics different than λHOL and our results should therefore generalize gracefully.

The VeriML computational layer

The computational core of VeriML is a functional language with call-by-value semantics and support for side-effects such as mutable references in the style of ML. This core calculus is extended with constructs that are tailored

to manipulation of a typed object language – which in the case of VeriML consists of the λHOL logic. Viewed generically, the definition and proofs of the computational layer expect the object language to be defined through a typing judgement of the form:

$$\Psi \vdash T : K$$

We also expect a separate judgement for patterns, of the form:

$$\Psi \Vdash^{\star}_{P} \Psi_{u} > T_{P} : K$$

which is read as 'in the context Ψ, assuming the unification variables Ψ_u, the pattern T_P has the type K'. We assume a pattern matching algorithm for matching a closed scrutinee T against a pattern T_P, resulting in a substitution for the unification variables of the pattern. Last, we assume that a standard simultaneous substitution theorem holds for both judgements described above; that pattern matching is decidable and deterministic; and that the pattern matching algorithm is sound and produces a valid substitution for all unification variables upon success.

In the case of λHOL, T contains two kinds of terms: open logical terms that are written as $[\Phi]\, t$ (where Φ explicitly records the context of variables that the open term t might refer to); and contexts of logical variables Φ. We refer to the class T as *extension terms*. The context Ψ stores variables to be instantiated with such extension terms: meta-variables, denoted as X, and context variables, denoted as ϕ. When using a metavariable X inside a logical term, we are allowed to use an explicit substitution for the variables Φ. We will present more details about extension terms, as well as the above judgements and the associated proofs later in this summary. We will now focus instead on the constructs of the computational language.

The ML core of the computational language supports the following features: higher-order functional programming; general recursion; algebraic data types; mutable references; polymorphism over types; and type constructors. Typing derivations are of the form $\Psi;\ \mathcal{M};\ \Gamma \vdash \mathcal{J}$, where Ψ is the extension variables context that will be used in the λHOL-related extensions; $\mathcal{M}$ is the store typing, i.e. a context assigning types to locations in the current store μ; and Γ is the context of computational variables, including type-level α and expression-level x variables. The operational semantics work on machine states of the form $(\ \mu\ ,\ e\ )$ which bundle the current store μ with the current expression e. We formulate the semantics through evaluation contexts $\mathcal{E}$. All of these definitions are entirely standard so we will not comment further on them.

The first important addition to the ML computational core are constructs for dependent functions and products over extension terms T. Abstraction over extension terms is denoted as $\lambda X : T.e$. It has the dependent function type $(X : T) \times \tau$. The type is dependent since the introduced term might be used as the type of another term. We can use this construct both to abstract over contexts, for functions that work over terms open in any context, as well as for abstracting over propositions, logical types and proofs. An example would be a function that receives a proposition plus a proof object for that proposition, both inhabiting the same arbitrary context; its type would be:

$$(\phi : ctx) \rightarrow (P : [\phi]\, Prop) \rightarrow (X : [\phi]\, P) \rightarrow \tau$$

Dependent products that package an extension term with an expression are introduced through the $\langle\ T,\ e\ \rangle$ construct and eliminated using $\mathsf{let}\ \langle\ X,\ x\ \rangle = e\ \mathsf{in}\ e'$; their type is denoted as $(X : T) \times \tau$. Especially for packages of proof objects with the unit type, we introduce the syntax (T). 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:

$$\mathsf{instantiate} : (\phi : ctx,\ T : [\phi]\, Type,\ P : [\phi,\ x : T]\, Prop,\ a : [\phi]\, T) \rightarrow \big([\phi]\, \forall x : T, P\big) \rightarrow \big([\phi]\, P/[id_{\phi},\ a]\big)$$

$$\mathsf{instantiate}\ \phi\ T\ P\ a\ \mathsf{pf}\ = \mathsf{let}\ \langle\ H\ \rangle = \mathsf{pf}\ \mathsf{in}\ \langle\ H\ a\ \rangle$$

The most important λHOL-related construct in VeriML is a pattern matching construct for looking into the structure of λHOL extension terms. We can use this construct to implement functions such as the auto automated prover suggested above; its formal version would be:

$$
\begin{array}{l}
\mathsf{auto} \quad : \quad (\phi : \mathit{ctx}) \to (P : [\phi]\,\mathit{Prop}) \to \mathsf{option}\,([\phi]\,\mathit{Prop}) \\
\mathsf{auto}\ \phi\ P \quad = \\
\quad \mathsf{holmatch}\ P\ \mathsf{with} \\
\qquad (Q : [\phi]\,\mathit{Prop},\ R : [\phi]\,\mathit{Prop}).[\phi]\,Q \wedge R \quad \mapsto \quad \mathsf{do}\ \ X \leftarrow \mathsf{auto}\ \phi\ ([\phi]\,Q)\,; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ Y \leftarrow \mathsf{auto}\ \phi\ ([\phi]\,R)\,; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \mathsf{return}\ \langle\, [\phi]\,\mathit{andI}\ X\ Y \,\rangle \\
\qquad \mid (Q : [\phi]\,\mathit{Prop},\ R : [\phi]\,\mathit{Prop}).[\phi]\,Q \to R \quad \mapsto \quad \mathsf{do}\ \ X \leftarrow \mathsf{auto}\,(\phi,\ Q)\,([\phi,\ Q]\,R) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \mathsf{return}\ \langle\, [\phi]\,\lambda(Q).X \,\rangle \\
\qquad \mid (P : [\phi]\,\mathit{Prop}).[\phi]\,P \qquad\qquad\qquad\qquad \mapsto \quad \mathsf{findHyp}\ \phi\ ([\phi]\,P)
\end{array}
$$

The variables in the parentheses represent the unification variables context Ψ_u, whereas the following extension term is the pattern. The findHyp function searches the context for a hypothesis matching its input proposition. It works by using the same pattern matching construct over contexts.

In the formal definition of VeriML we give a very simple pattern matching construct that just matches a term against one pattern. More complicated pattern matching constructs are derived forms of this basic construct and are presented in **Section 6.2** of the dissertation. The basic construct is written as:

$$\mathsf{holmatch}\ T\ \mathsf{return}\ V : K.\tau\ \mathsf{with}\ \Psi_u.T_P \mapsto e$$

The term T represents the scrutinee, Ψ_u the unification context, T_P the pattern and e the body of the match. The return clause specifies what the return type of the construct will be, using V to refer to T. We omit this clause when it is directly inferrable from context. The typing rule for this construct is:

$$\frac{\Psi \vdash T : K \qquad \Psi,\ V : K;\ \Gamma \vdash \tau : \star \qquad \Psi \mid^{\star}_{P}\ \Psi_u > T_P : K \qquad \Psi,\ \Psi_u;\ \mathcal{M};\ \Gamma \vdash e' : \tau[T_P/V]}{\Psi;\ \mathcal{M};\ \Gamma \vdash \begin{array}{l}\mathsf{holmatch}\ T\ \mathsf{return}\ V : K.\tau \\ \mathsf{with}\ \ \Psi_u.\ T_P \mapsto e'\end{array} : \tau[T/V] + \mathsf{unit}}\ \textsc{CExp-HolMatch}$$

As this typing rule demonstrates, pattern matching is dependently typed, as the pattern T_P refines the expected type τ of the body of the branch. The return type of the construct is an option type wrapping $\tau[T/V]$ in order to capture the possibility of pattern match failure.

The operational semantics of this construct are as follows:

$$\frac{T_P \sim T = \sigma_\Psi}{(\ \mu\ ,\ \mathsf{holmatch}\ T\ \mathsf{with}\ \Psi_u.T_P \mapsto e'\ ) \longrightarrow (\ \mu\ ,\ \mathsf{inj}_1\,(e' \cdot \sigma_\Psi)\ )}\ \textsc{Op-HolMatch}$$

$$\frac{T_P \sim T = \mathsf{fail}}{(\ \mu\ ,\ \mathsf{holmatch}\ T\ \mathsf{with}\ \Psi_u.T_P \mapsto e'\ ) \longrightarrow (\ \mu\ ,\ \mathsf{inj}_2\,()\ )}\ \textsc{Op-HolNoMatch}$$

They are defined through the pattern matching algorithm $T_P \sim T$ we assumed above.

Section 6.1 of the dissertation presents the main metatheoretic proofs about VeriML, culminating in a *type-safety* theorem. The usual informal description of type-safety is that well-typed programs do not go wrong. A special case of this theorem is particularly revealing about what this theorem entails for VeriML: the case of programs e typed as (P), where P is a proposition. Values $v = \langle\, \pi \,\rangle$ having a type of this form include a proof object π proving P. Expressions of such a type are arbitrary programs that may use all VeriML features (functions, λHOL pattern matching, general recursion, mutable references etc.) in order to produce a proof object; we refer to these expressions as proof expressions or proof scripts. Type safety guarantees that *if evaluation of a proof expression of type* (P) *terminates, then it will result in a value* $\langle\, \pi \,\rangle$ *where* π *is a valid proof object for the proposition* P. Thus we do not need to check the proof object produced by a proof expression again using a proof checker. Another way to view the same result is that we cannot evade the λHOL proof checker embedded in the VeriML type system – we cannot cast a value of a different type into a proof object type (P) only to discover at runtime that this value does not include a

proper proof object. Of course the type safety theorem is much more general than just the case of proof expressions, establishing that an expression of *any* type that terminates evalutes to a value of the same type. The formal statement of this theorem is as follows:

$$\frac{\bullet;\, \mathcal{M};\, \bullet \vdash e : \tau \qquad \mu \sim \mathcal{M} \qquad e \neq v}{\exists \mu', \mathcal{M}', e'.(\quad (\,\mu\,,\, e\,) \longrightarrow (\,\mu'\,,\, e'\,) \;\wedge\; \bullet;\, \mathcal{M}';\, \bullet \vdash e' : \tau \;\wedge\; \mu' \sim \mathcal{M}' \quad)}$$

The relation $\mu \sim \mathcal{M}$ means that the store μ matches exactly the store typing $\mathcal{M}$. As is standard, we split this proof into two main lemmas: preservation and progress.

$$\frac{\bullet;\, \mathcal{M};\, \bullet \vdash e : \tau \qquad (\,\mu\,,\, e\,) \longrightarrow (\,\mu'\,,\, e'\,) \qquad \mu \sim \mathcal{M}}{\exists \mathcal{M}'.(\quad \bullet;\, \mathcal{M}';\, \bullet \vdash e' : \tau \;\wedge\; \mathcal{M} \subseteq \mathcal{M}' \;\wedge\; \mu' \sim \mathcal{M}' \quad)}\ \textbf{Preservation}$$

$$\frac{\bullet;\, \mathcal{M};\, \bullet \vdash e : \tau \qquad \mu \sim \mathcal{M} \qquad e \neq v}{\exists \mu', e'.(\,\mu\,,\, e\,) \longrightarrow (\,\mu'\,,\, e'\,)}\ \textbf{Progress}$$

The bulk of the proof consists exactly of the proofs about λHOL that we have assumed, as is evident for example from the pattern matching construct. We directly use these proofs in order to prove type-safety for the new constructs of VeriML; and type-safety for the constructs of the ML core entirely follows standard practice.

Staging

We mentioned earlier that a unique characteristic of VeriML is the possibility to evaluate certain tactic calls statically, at the time of definition of a new tactic – rather than at runtime. We use this in order to 'fill in' proof objects in the new tactic through existing automation code; furthermore, the proof objects are created once and for all when the tactic is defined and not every time the tactic is invoked.

This support is the combination of two features: a *logic-level feature* called *collapsing*, which is the transformation of λHOL open terms with respect to the extension context Ψ to equivalent closed terms; and a *computational-level feature*, namely the ability to evaluate subexpressions of an expression during a distinct evaluation stage prior to normal execution. This latter feature is supported by extending the VeriML computational language with a *staging construct*; it is presented in **Section 6.2** of the dissertation, along with the proof that progress and preservation continue to hold under this extension. This staging construct is called letstatic and can be seen as a user-directed generalization of constant propagation. It is written as letstatic $x = e$ in e', where x is a static variable and e can only mention static variables.

An example where such a feature is useful is in the definition of rewriting tactics to be used as part of conversion. Consider the example of a tactic that simplifies terms such as $x + (succ\ y)$ into $succ\ (x + y)$, where *succ* is the successor constructor of natural numbers. A sketch of this function is shown in Figure 1.

While G_1 is the interesting part of the tactic and thus we certainly need to prove its associated lemma, in other cases such as G_2, the corresponding lemma is uninteresting and tedious to state and prove. Rather than proving it by hand, we could use the congruence closure conversion rule to solve this trivial obligation directly. Yet we can do even better, using the letstatic construct. We can evaluate the call to decideEqual statically, during stage one interpretation. Thus we will know at the time that plusRewriter1 is defined whether the call succeeded, allowing us to catch potential errors earlier. Also, the call will be replaced by a concrete value and thus later invocations of plusRewriter1 will not have to perform the proof search again. In order to use letstatic, we need to avoid mentioning any of the metavariables that are bound during runtime, like X, Y, and E'. This is done by specifying an appropriate environment in the call to decideEqual:

```
type rewriterT = (φ : ctx, T : Type, E : T) → (E' : T) × (E = E')

plusRewriter1 : rewriterT → rewriterT
plusRewriter1 recurse φ T E =
  holmatch E with
  | X + Y ↦
    let ⟨ Y', H1 : Y = Y' ⟩ = recurse φ Y in
    let ⟨ E', H2 : X + Y' = E' ⟩ =
      holmatch Y' with
      | succ Y' ↦ ⟨ succ (X + Y'), G1 : X + succ Y' = succ (X + Y') ⟩ | ⋯
    in
    ⟨ E', G2 : X + Y = E' ⟩
```

Figure 1: Simplification of natural number expressions

$$
\begin{aligned}
G_2 = \ & \mathsf{letstatic}\ \langle\, H \,\rangle = \\
& \quad \mathsf{let}\ \Phi'' = [x, y, y', e' : Nat, h_1 : y = y', h_2 : x + y' = e'] \ \mathsf{in} \\
& \quad \mathsf{decideEqual}\ \Phi''\ (x + y)\ t' \\
& \mathsf{in}\ \langle\, [\phi]\, H / [X/id_\phi, Y/id_\phi, Y'/id_\phi, E'/id_\phi, H_1/id_\phi, H_2/id_\phi] \,\rangle
\end{aligned}
$$

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. A surface-level syntactic extension together with type inference allow us to omit all of the details, writing a concise expression instead: {{decideEqual}}.

We present more details about the above use case in Section 6.2; we also show how the same technique can be used for dependently-typed programming within VeriML in Section 6.3. We will now briefly present the formal details of the letstatic construct. Its typing rule reflects the fact that only static variables are allowed in staged expressions e by removing non-static variables from the context through the context limiting operation $\Gamma|_{\text{static}}$:

$$
\frac{\bullet;\ \mathcal{M};\ \Gamma|_{\text{static}} \vdash e : \tau \qquad \Psi;\ \mathcal{M};\ \Gamma, x :_s \tau \vdash e' : \tau}{\Psi;\ \mathcal{M};\ \Gamma \vdash \mathsf{letstatic}\ x\ =\ e\ \mathsf{in}\ e' : \tau}\ \textsc{CExp-LetStatic}
$$

The operational semantics are adapted as follows: we define a new small-step reduction relation $\longrightarrow_s$ between machine states, capturing the static evaluation stage where expressions inside letstatic are evaluated. If this stage terminates, it yields a residual expression e_d which is then evaluated using the normal reduction relation $\longrightarrow$. With respect to the first stage, residual expressions can thus be viewed as *values*. Furthermore, we define static binding evaluation contexts $\mathcal{S}$: these are the equivalent of evaluation contexts $\mathcal{E}$ for static evaluation. The key difference is that static binding evaluation contexts can also go under non-static binders. These binders do not influence static evaluation as we cannot refer to their variables due to the $\Gamma|_{\text{static}}$ operation.

Proof erasure

Thanks to the type safety of VeriML we know that all yielded proof objects are guaranteed to be valid with respect to their statically-determined type. There are many situations where we are interested in the exact structure of the proof object. For example, we might want to ship the proof object to a third party for independent verification. In most situations though, we are interested merely in the existence of the proof object. In **Section 6.3** of the dissertation we will show that in this case, we can evalute the original VeriML expression without producing proof objects at any intermediate step but still maintain the same guarantees about the existence of proof objects. This is done through the *proof erasure* semantics mentioned earlier.

More formally, we define a type-directed translation from typing derivations of expressions to expressions with all proofs erased $[\![\Psi;\ \mathcal{M};\ \Gamma \vdash e : \tau]\!]_E = e'$. We then show a strict bisimulation between the normal semantics and the semantics where the erasure function has first been applied – that is, that every evaluation step under the normal semantics $(\ \mu\ ,\ e\ ) \longrightarrow (\ \mu'\ ,\ e'\ )$ is simulated by a step like $\left(\ [\![\mu]\!]_E\ ,\ [\![e]\!]_E\ \right) \longrightarrow \left(\ [\![\mu']\!]_E\ ,\ [\![e']\!]_E\ \right)$, and also that the inverse holds. The essential reason why this bisimulation exists is that the pattern typing rules $\Psi \vdash^{\pm}_{p} \Psi_u > T_P : K$ contain a sort restriction so that patterns for individual proof object constructors do not exist. Thus the pattern matching construct of VeriML cannot be used to inspect deep into the structure of proof objects. Since pattern matching is the only computational construct available for looking into the structure of logical terms, this restriction is enough so that proof objects will not influence the runtime behavior of VeriML expressions.

Implementation and examples

We have completed a prototype implementation of VeriML that includes all the features we have seen so far as well as a number of other features that are practically essential, such as a type inference mechanism. Our prototype also includes a complete implementation of the λHOL logic extended with inductive definitions in the style of Coq. The prototype is programmed in the OCaml programming language [22] and consists of about 10k lines of code. It is freely available from `http://www.cs.yale.edu/homes/stampoulis/`. We describe this implementation in Chapter 8 of the dissertation.

Evaluation of VeriML programs is done through translation of well-typed VeriML expressions to OCaml expressions. The resulting OCaml expressions can then be treated as normal OCaml programs and compiled using the normal OCaml compiler. This combination can thus be viewed as a compiler for VeriML programs and performs at least one order of magnitude better than an interpreter-based implementation of VeriML we had completed earlier.

Our implementation also includes a VeriML toplevel – an interactive read-eval-print-loop (REPL) environment as offerred by most functional languages. It works by integrating the VeriML-to-OCaml translator with the OCaml just-in-time interpreter and is suitable for experimentation with the language or as a development aid. It is also suitable for use as a proof assistant. When defining a new VeriML expression – be it a tactic, a proof script or another kind of computational expression – the VeriML type checker informs us of any type errors. By leaving certain parts of such expressions undetermined, such as a part of a proof script that we have not completed yet, we can use the type information returned by the type checker in order to proceed. Furthermore, we are also informed if a statically-evaluated proof expression fails. We have created a wrapper for the toplevel for the popular Proof General frontend to proof assistants, which simplifies this style of dialog-based development.

We have implemented a type inference engine both for the logic and the computational layer, allowing us to omit typing annotations in constructs such as the return clause in holmatch, implicit arguments to functions, instantiations of type parameters etc. Also, we have special syntactic support for limiting the annotation burden associated with the use of extension terms, allowing us to omit the context part Φ in open terms such as $[\Phi]\ t$ and the explicit substitution σ in uses of meta-variables X/σ. This is done by maintaining an ambient Φ context which most contextual terms refer to; this context is denoted as @. It is implicitly updated by operations such as abstraction over contexts, but also through explicit user-level constructs, such as $\nu x : t$ in e for introducing a new variable. Last, we have specials surface-level syntax for tactics in order to simplify the development of proof scripts that use them.

Thanks to these features, proof scripts and tactics in VeriML have relatively limited typing annotations and look similar to their untyped counterparts. We have developed a number of examples using the prototype, consisting of about 1.5k lines of VeriML code. We present these in **Chapter 9** of the dissertation. We follow the ideas presented earlier for building a conversion rule that supports β-equivalence, congruence closure and arithmetic simplifications in successive layers. Such extensions to the conversion rule have required significant engineering effort in the past [e.g. 20]; their implementation in current proof assistants makes use of techniques such as proof by reflection and translation validation. The VeriML design allows us to express them in a natural style. We also present a first-order automated theorem prover which will be used to simplify some of the proofs involved.

As an example, we show the main function that performs rewriting of arithmetic expressions into a canonical form below.

```
let rewriter_arithsimpl : rewriter_module_t =
  fun recursive {φ T} e ⇒
  holmatch @e with
    @e1 + e2 ↦
        let < l0, pf1 > = nat_to_monolist @e1+e2 in
        let < l1, pf2 > = sort_list cmp @l0 in
        let < l2, pf3 > = factorize_monolist @l1 in
        < @sumlist l2, {{ Cut @permutation_sum l0 l1 in Auto }} >
  | @e ↦ < @e , StaticEqual > ;;
```

This function first converts the input natural number into a list of monomials; then, it sorts the list based on the hash value of the variable of each monomial; it proceeds to factorize adjacent monomials on the same variable; and it returns the sum of the resulting list as the canonical form e'. Each individual function returns an associated proof. For example, `sort_list` proves that its result is a permutation of its input. These proofs are combined to prove that $e = e'$ through the proof script `Cut ...`. It is evaluated statically, at the time of the definition of the tactic thanks to the annotation $\{\{\cdot\}\}$. The lemma `permutation_sum` shows that permutations of the same list have equal sums; the rest of the proof is carried out through the `Auto` prover.

The base λHOL logic

We now return to the definition of the λHOL logic and the question of how λHOL is made to satisfy the assumptions that the computational layer expects, with regards to the substitution theorems and pattern matching algorithm.

Chapter 3 presents the base λHOL logic in three different formulations. First, we give a stratified view of λHOL, with the separate judgements for classifying terms of different classes as mentioned earlier – e.g. a judgement $\Phi \vdash \pi : P$ is used for specifying valid proofs and a judgement $\Phi \vdash d : \mathcal{K}$ is used for domain objects. We show the equality rules that need to be added to such a logic in order to get a logic with explicit equality witnesses with the same expressiveness as a logic with the $\beta\iota$-conversion rule. We present an equivalent formulation of λHOL as a PTS, where all classes of terms are collapsed into the class t of logical terms and the logic is defined through a single judgement of the form $\Phi \vdash t : t'$.

Last, we define another equivalent formulation of λHOL which uses a concrete representation technique for variables, as is usually done in developments involving languages with binding inside proof assistants. We have found that using a concrete variable representation technique is advantageous for our development, even though our proofs are done on paper. The main reason is that the addition of contextual terms, contextual variables and polymorphic contexts that VeriML requires introduces a number of operations which are hard to define in the presence of named variables, as they trigger complex sets of α-renamings. Using a concrete representation, these operations and the renamings that are happening are made explicit. Furthermore, we use the same concrete representation in our implementation of VeriML itself, so the gap between the formal development and the actually implemented code is smaller, increasing our confidence in the overall soundness of the system.

The concrete representation technique that we use is a novel, to the best of our knowledge, combination of the techniques of de Bruijn indices, de Bruijn levels and the locally nameless approach. The former two techniques replace all variables by numbers, whereas the locally nameless approach introduces a distinction between bound and free variables and handles them differently. We merge these approaches in a new technique that we call *hybrid deBruijn variables*: following the locally nameless approach, we separate free variables from bound variables and use deBruijn indices for bound variables; but also, we use deBruijn levels for free variables instead of names. In this way, all α-equivalent terms are identified. Furthermore, terms preserve their identity under additions to the end of the free variables context. Such additions are a frequent operation in VeriML, so this representation is advantageous from an implementation point of view. This representation requires two main operations for dealing with binders: a *freshening* operation used when crossing a binder, to change the just-bound variable b_0 needs to be converted into a fresh free variable; and the inverse operation called *binding*, used when we re-introduce a binder, capturing the last free variable as a bound variable.

With this representation, applying a simultaneous substitution to a term is a simple structural recursion, the only interesting case being the replacement of the i-th free variable with the i-th term of the substitution. We denote simultaneous substitutions as σ and the application of a substitution to a term as $t \cdot \sigma$. Substitutions are typed through judgements of the form $\Phi \vdash \sigma : \Phi'$, representing the fact that σ assigns terms to every variable of the context Φ' and those terms contain variables from the context Φ.

The main result of this chapter is that the substitution theorem and the weakening theorem hold for the version of λHOL formulated with hybrid deBruijn variables:

$$\frac{\Phi \vdash t : t' \qquad \Phi' \vdash \sigma : \Phi}{\Phi' \vdash t \cdot \sigma : t' \cdot \sigma}\ \textsc{Substitution Theorem}$$

$$\frac{\Phi \vdash t : t' \qquad \Phi' = \Phi, t_1, t_2, \cdots, t_n \qquad \vdash \Phi' \text{ wf}}{\Phi' \vdash t : t'}\ \textsc{Weakening}$$

Meta-variables and parametric contexts for λHOL

In **Chapter 4**, we present two extensions to the λHOL logic in order to support the actual logical manipulation constructs we need in computational layer of VeriML. The two extensions essentially *internalize the substitution and weakening principles within the logic*; we can therefore view them as syntactic rather than semantic extensions. The first extension corresponds to adding meta-variables to λHOL, which are used so that VeriML can manipulate *open* logical terms inhabiting specific contexts. The second extension adds context variables, which are necessary so that we can manipulate contexts themselves, as well as open terms that inhabit *any* context – through combination with the meta-variables support. These extensions are presented in two steps in the interests of presenation. In both cases we present the metatheoretic proofs in detail.

A *meta-variable* is a special kind of variable which is able to 'capture' variables from the current context in a safe way, when it is substituted by a term t. The type of this meta-variable needs to contain both the information about the expected environment where t will make sense in addition to the type of the term. Types of metavariables are therefore *contextual terms*: a package of a context Φ together with a term t, which we write as $[\Phi]\, t$ and denote as T in our development development. Instantiations of meta-variables are contextual terms too, in order to show clearly the environment of variables that can be used in the contained open term; we thus refer to meta-variables alternatively as contextual variables.

We extend λHOL with a context of meta-variables denoted as Ψ, and add a rule for using a meta-variable of the following form, drawing from contextual modal type theory:

$$\frac{X : [\Phi'] \, t \in \Psi \qquad \Phi \vdash \sigma : \Phi'}{\Psi;\, \Phi \vdash X/\sigma : t \cdot \sigma}$$

Assuming that X is instantiated with the contextual term $[\Phi'] \, t'$, this typing rule essentially requires a mapping between the variables that t' refers to and terms that make sense in the context Φ where the metavariable is being used. This mapping takes the form of the explicit substitution σ; in the most common case, this will just be the identity substitution id_Φ. We also add a new notion of substitutions for metacontexts Ψ: these are lists of contextual terms instantiating all metavariables in Ψ. We call them *metasubstitutions* and denote them as σ_Ψ.

One important metatheoretic result of this section is that the syntactic theorems that we proved earlier continue to hold after the extension. In order to avoid redoing the proofs for the lemmas we have already proved, we follow an approach towards extending our existing proofs that resembles the open recursion solution to the expression problem. Yet the most important metatheoretic result of this section is that similar theorems hold for meta-substitutions as well. Formally, the main result of this section is the following meta-substitution theorem:

$$\frac{\Psi;\, \Phi \vdash t : t' \qquad \Psi' \vdash \sigma_\Psi : \Psi}{\Psi';\, \Phi \cdot \sigma_\Psi \vdash t \cdot \sigma_\Psi : t' \cdot \sigma_\Psi} \text{ METASUBSTITUTION THEOREM}$$

In Section 4.2 we add a further extension in order to account for parametric contexts in addition to meta-variables. The rationale behind this extension is as follows: most functions in VeriML work with logical terms that live in an arbitrary context. This is for example true for automated provers such as Auto as mentioned earlier, which need to use recursion to prove propositions in extended contexts. Thus, even if the original context is closed, recursive calls work over terms living in extensions of the context; and the prover needs to handle all such extensions – it needs to be able to handle all possible contexts. Parametric contexts can include context variables (denoted as ϕ) that can be instantiated with an arbitrary context. We can see context variables as placeholders for applying the weakening lemma. Thus this extension internalizes the weakening lemma within the λHOL calculus, just as the extension with metavariables internalizes the substitution theorem.

The main ideas of the extension are the following:

1. Both context variables and meta-variables are typed through the same environment and are instantiated through the same substitutions, instead of having separate environments and substitutions for context- and meta-variables.

2. The deBruijn levels used for normal free variables are generalized to *parametric deBruijn levels*: instead of being constant numbers they are sums involving variables $|\phi|$ standing for the length of as-yet-unspecified contexts.

Based on these ideas, the definitions of operations such as substitution application, as well as our main proofs are carried out through structural recursion and are conceptually clear; this is one of the main technical contributions of the dissertation in the domain of contextual modal type theory. Parametric deBruijn levels allow for a clean way to perform the necessary α-renamings when instantiating a parametric context with a concrete context, by simply substituting the length variables for the concrete length. The justification for the first key idea noted above is technically more subtle: by separating meta-variables from context variables into different contexts, the relative order of their introduction is lost. This leads to complications when a meta-variable depends on a parametric context which depends on another meta-variable itself and requires various side-conditions to the statements of lemmas that add significant technical complexity.

Through this extension, the context Ψ now includes both meta-variables and context variables – called extension variables; the extension substitution σ_Ψ now includes instantiations for both of these kinds of variables. The instantiations of extension variables are *extension terms* T, typed through extension types K. In the case of metavariables, an instantiation is a contextual term $[\Phi] \, t$ with contextual type $[\Phi] \, t'$. In the case of a context variable ϕ, an instantiation is a context Φ' that extends the prefix Φ, denoted as $[\Phi] \, \Phi'$. Its type is denoted as $[\Phi] \, ctx$.

The main result of this section is the following theorem for applying an extension substitution to an extension term:

$$\frac{\Psi \vdash T : K \qquad \Psi' \vdash \sigma_\Psi : \Psi}{\Psi' \vdash T \cdot \sigma_\Psi : K \cdot \sigma_\Psi}\ \textsc{Extension Substitution Theorem}$$

Last, in Section 4.3 we show how to utilize metavariables in order to allow a restricted form of polymorphism over kinds in λHOL. This addition is useful for polymorphic types such as lists and for typing induction principles. Instead of adding actual abstraction over the *Type* sort which would render the logic unsound per Girard's paradox, we introduce a signature of constant schemata Σ (instead of constants). Each constant schema is represented through a meta-variable; the rule presented earlier for using a meta-variable with an explicit substitution corresponds to an instantiation of such a schema.

Pattern matching for λHOL

In **Chapter 5** we present the main operation through which λHOL terms are manipulated in VeriML: pattern matching. We first describe the procedure and metatheoretic proofs for matching a scrutinee against a pattern in Section 5.1. In Section 5.2, we present the collapsing approach that enables us to use the same pattern matching procedure when the scrutinee is not closed with respect to extension variables; this is the main technical justification behind staging calls to tactics.

Informally we can say that a pattern is the 'skeleton' of a term, where some parts are specified and some parts are missing. We name each missing subterm using *unification variables*; as these variables might occur under the binding constructs of λHOL, they are represented as meta-variables in order to capture their environment information. The substitution returned from pattern matching as being composed from contextual terms to be substituted for these (unification) meta-variables. The situation for matching contexts against patterns that include context unification variables is similar. Thus we view patterns as a special kind of extension terms where the extension variables used are viewed as unification variables. We place further restrictions on what terms are allowed as patterns, resulting in a new typing judgement that we denote as $\Psi \vdash_p T : K$. Based on these, we formulate pattern matching for λHOL as follows. Assuming a pattern T_p and an extension term T (the *scrutinee*) with the following typing:

$$\Psi_u \vdash_p T_p : K \qquad \text{and} \qquad \bullet \vdash T : K$$

where the extension context Ψ_u describes the unification variables used in T_p, pattern matching is a procedure which decides whether a substitution σ_Ψ exists so that:

$$\bullet \vdash \sigma_\Psi : \Psi_u \qquad \text{and} \qquad T_p \cdot \sigma_\Psi = T$$

We proceed as follows.

1. First, we define the typing judgement for patterns $\Psi \vdash_p T : K$. This is mostly identical to the normal typing for terms, save for certain restrictions so that pattern matching is decidable and deterministic.

2. Then, we define an operation to extract the *relevant variables* out of typing derivations; it works by isolating the *partial context* that gets used in a given derivation. This is useful for two reasons: first, to ensure that all the defined unification variables get used in a pattern; and second, in order to get the proper induction principle that we need for the pattern matching theorem that we prove.

3. Using this operation, we define an even more restrictive typing judgement for patterns, which requires all unification variables defined to be relevant. A sketch of its formal definition is as follows:

$$\frac{\Psi, \Psi_u \vdash_p T_p : K \qquad \mathit{relevant}\left(\Psi, \Psi_u \vdash_p T_p : K\right) = \cdots, \Psi_u}{\Psi \vdash^+_p \Psi_u > T_p : K}$$

We prove a substitution theorem for this rule, with the following formal statement:

$$\frac{\Psi \vdash^{\star}_{p} \Psi_u > T_p : K \qquad \Psi' \vdash \sigma_\Psi : \Psi}{\Psi' \vdash^{\star}_{p} \Psi_u \cdot \sigma_\Psi > T_p \cdot \sigma_\Psi : K \cdot \sigma_\Psi} \textsc{Pattern Substitution Lemma}$$

This theorem extends the extension substitution theorem so as to apply to the case of patterns as well.

4. Based on these, we prove the fact that pattern matching is decidable and deterministic. We carry out the proof in a constructive manner; its computational counterpart is the pattern matching algorithm that we use. Therefore the pattern matching algorithm is sound and complete directly by construction. We write $T_p \sim T = \sigma_\Psi$ for an execution of this algorithm between the pattern T_p and the scrutinee T, when it results in a substitution σ_Ψ. Formally, the statements of the main theorems we prove are as follows.

$$\frac{\bullet \vdash^{\star}_{p} \Psi_u > T_p : K \qquad \bullet \vdash T : K}{\exists^{\leq 1} \sigma_\Psi . (\quad \bullet \vdash \sigma_\Psi : \Psi_u \qquad T_p \cdot \sigma_\Psi = T \quad)} \begin{array}{l}\textsc{Pattern matching} \\ \textsc{determinism and decidability}\end{array}$$

$$\frac{\bullet \vdash^{\star}_{p} \Psi_u > T_p : K \qquad \bullet \vdash T : K \qquad T_p \sim T = \sigma_\Psi}{\bullet \vdash \sigma_\Psi : \Psi_u \qquad T_p \cdot \sigma_\Psi = T} \begin{array}{l}\textsc{Pattern matching} \\ \textsc{algorithm soundness}\end{array}$$

The introduction of partial contexts and the corresponding notion of partial substitutions that we use in order to carry out the proof of pattern matching is a technical contribution of our work. It reconciles the concrete variable approach that we follow with pattern matching and also allows to carry out the majority of the proof using structural induction.

In Section 5.2, we give the definition of the collapsing transformation which is used when making staged calls to tactics through the $\{\{\cdot\}\}$ construct. More formally, the issue is the following: the pattern matching methodology as presented does not apply when the scrutinee is not closed with respect to the Ψ context. We partially address this issue by showing that under certain restrictions about the current extension context Ψ and its usage, we can *collapse* terms from Ψ to an empty extension context. A substitution exists that then transforms the collapsed terms into terms of the original Ψ context. That is, we can get an equivalent term that is closed with respect to the extension context and use our normal pattern matching procedure on that.

We thus prove the following theorem for contextual terms:

$$\frac{\Psi;\, \Phi \vdash t : t' \qquad \mathit{collapsable}\,(\Psi \vdash [\Phi]\, t : [\Phi]\, t')}{\exists t_\downarrow, t'_\downarrow, \Phi_\downarrow, \sigma . (\quad \bullet;\, \Phi_\downarrow \vdash t_\downarrow : t'_\downarrow \qquad \Psi;\, \Phi \vdash \sigma : \Phi_\downarrow \qquad t_\downarrow \cdot \sigma = t \qquad t'_\downarrow \cdot \sigma = t' \quad)}$$

Combined with the pattern matching algorithm given in the previous section, which applies when $\Psi = \bullet$, this gives us a way to perform pattern matching even when Ψ is not empty. The restrictions that the "collapsable" operation denotes are satisfied in most situations that arise practically in our examples: all involved variable contexts in the term t should be prefixes of one global Φ^* context, and only identity substitutions can be used to instantiate metavariables.

5 Conclusion

In conclusion, this dissertation presents the design, metatheory and implementation of VeriML: a programming language that enables the development of verified tactics, using an expressive programming model and convenient constructs tailored to this task. We achieve expressiveness by including the side-effectful, general-purpose programming model of ML as part of the language; convenience by adding first-class support for introducing proofs, propositions and other logical terms as well as pattern matching over them – the exact constructs that form the main building blocks of tactics in current proof assistants. The verification of tactics is done in a light-weight manner, so that it does

not become a bottleneck in itself. It is the direct product of two main features. First, the rich type system of the language that keeps information about the logical terms that are manipulated; and second, a staging mechanism which is informed by the typing information and allows the use of existing automation in order to minimize the proof burden associated with developing new tactics. The use of the type system also leads to a principled programming style for tactics, making them more composable and maintainable.

We show that various features of the architecture of modern proof assistants can be directly recovered and generalized through the VeriML design, rendering the VeriML language as a proof assistant in itself. We also show how the conversion rule that is part of the trusted core of modern proof assistants, can be removed from the core and programmed instead in VeriML, thus allowing sophisticated user extensions while maintaining logical soundness.

References

[1] 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 8.4), 2012.

[2] L.C. Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer, 1994.

[3] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Conference record of the 33rd ACM SIGPLAN Symposium on Principles of Programming Languages*, page 54. ACM, 2006.

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

[5] A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. In *Proceedings of the 37th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 261–274. ACM, 2010.

[6] A. Asperti and C. Sacerdoti Coen. Some Considerations on the Usability of Interactive Provers. *Intelligent Computer Mathematics*, pages 147–156, 2010.

[7] 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.

[8] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2–3), 1988.

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

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

[11] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer, 2002.

[12] 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.

[13] M. Kaufmann and J. Strother Moore. ACL2: An industrial strength version of Nqthm. In *Computer Assurance, 1996. COMPASS'96,'Systems Integrity. Software Safety. Process Security'. Proceedings of the Eleventh Annual Conference on*, pages 23–34. IEEE, 1996.

[14] M.J. Gordon, R. Milner, and C.P. Wadsworth. Edinburgh LCF: a mechanized logic of computation. *Springer-Verlag Berlin*, 10:11–25, 1979.

[15] 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.

[16] A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3), 2008.

[17] D. Delahaye. A proof dedicated meta-language. *Electronic Notes in Theoretical Computer Science*, 70(2):96–109, 2002.

[18] 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.

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

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

[21] H. Geuvers and F. Wiedijk. A logical framework with explicit conversions. In *C. Schürmann (ed.), Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM 2004)*, page 32. Elsevier, 2004.

[22] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Remy, J. Vouillon, et al. The OCAML language (version 3.12. 0), 2010.