VeriML: Type-safe computation with terms of higher-order logic

Antonis Stampoulis   Zhong Shao

Department of Computer Science, Yale University

Dependently Typed Programming 2010
July 9-10, Edinburgh, UK

Introduction

Goal of this work

Design a language that combines general-purpose programming constructs with first-class support for manipulation of propositions and proof objects.

Motivation

Provide good language support for writing domain-specific tactics and decision procedures, to be used as part of large-scale proof development.

Motivation

In large proof developments in proof assistants like Coq and Isabelle, the user needs to construct domain-specific tactics and decision procedures to greatly reduce manual proving effort.

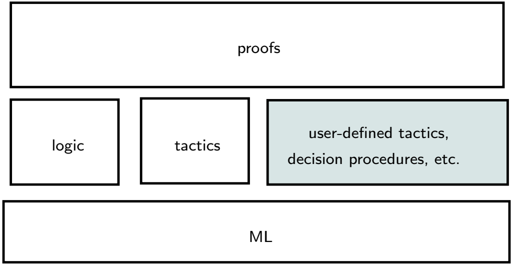
- Evidenced in the YNot project: verifying imperative programs with relatively small manual effort thanks to domain-specific tactics.
- Another example: different parts of an OS kernel require different abstraction levels; therefore should be verified with different program logics. Want automated provers for each one of them!

Developing tactics

Current language support is lacking. Available choices:

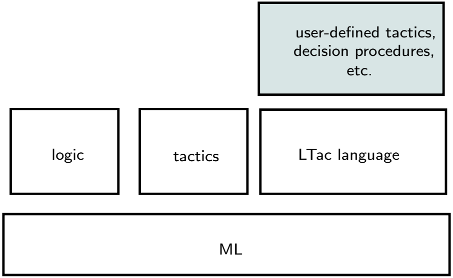
- Write them in ML (LCF family, Coq, Isabelle)
- Use LTac (Coq)
- Use proof-by-reflection

Developing tactics in ML



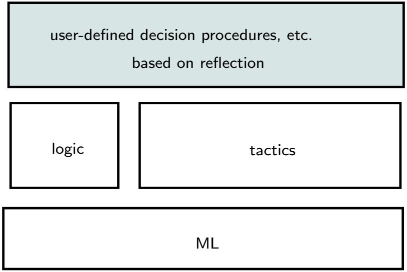
users have to know internals of proof assistant
no information at the type-level about logical terms
programming expressivity of ML (general recursion, imperative data structures)

Developing tactics in LTac



support for pattern matching over logical terms
general recursion allowed
completely untyped language, no static guarantees of safety
complex data-structures or imperative features not supported
binding not always handled correctly

Developing tactics using proof-by-reflection



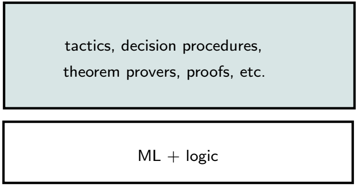
type-safe support for fragment of logic

strong static guarantees

requires use of mix of languages, tedious encoding (esp. when binding is involved)

limited programming model: no general recursion or imperative features

Our approach: VeriML



explicit, type-safe support for logic
full programming model of ML

Language design overview

Basic features of ML-like language

- General recursion
- Higher-order functions
- Algebraic datatypes (lists, trees, etc.)
- References, arrays

Logic-specific features

- Dependent types for terms of higher-order logic
- Pattern-matching construct for logical terms

Language design overview

Similar to languages for computation with LF terms (Delphin and Beluga). But:

- Instead of LF we use a higher-order logic that is a type theory modeled after CIC – thus it includes a notion of computation and terms are identified up to it.
- Terms in the computational language are not meant to be seen as meta-logical proofs.

The higher-order logic that we use: $\lambda\mathrm{HOL}^{\mathrm{ind}}$

A combination of a simple higher-order logic with proof objects (λ_{HOL}) with some features of the Calculus of Inductive Constructions.

- Propositions and inductive data types living in same universe
- No dependent types, no ω-quantification
- Total functions over inductive data types
- Inductive predicates for logical connectives and relations
- Explicit proof objects classified by propositions
- Logical terms are viewed up to evaluation of total functions ($\beta\iota$-equivalence)

The higher-order logic that we use

$$
\begin{aligned}
{}_{\text{kinds}}(K) &::= \text{inductive types} : Nat, List, \cdots \mid Prop \\
&\quad \mid K_1 \to K_2 \\
{}_{\text{dom. of discourse}}(d) &::= \text{terms of inductive types} : \\
&\quad x, zero, succ(d), nil, cons(d, d), \cdots \\
&\quad \mid \text{total functions between inductive types} \\
{}_{\text{propositions}}(P) &::= P_1 \to P_2 \mid \forall x : K.P \\
&\quad \mid \text{inductively defined predicates} : \leq, \cdots \\
&\quad \mid \text{inductively defined connectives} : \wedge, \vee, \neg, \cdots \\
{}_{\text{proof objects}}(\pi) &::= \lambda x : P.\pi \mid \pi\ \pi' \mid \lambda x : K.\pi \mid \pi\ d \\
&\quad \mid \text{elimination principles for inductive definitions} \\
{}_{\text{HOL terms}}(t) &::= K \mid d \mid P \mid \pi
\end{aligned}
$$

Typing: $\boxed{\Phi \vdash t : t'}$

Why we chose this logic

- Simple, "uncontroversial" common core between most proof assistants
- Easy to extend if needed
- Simple metatheory
- Equivalence up to computation reduces proof object size
- Maintains most of the complexities of theories like CIC, such that our results can be extended to them

Example: Propositional tautologies prover

Example: tauto

A simple automated prover for propositional tautologies.
Given a proposition, attempt to construct a proof object for it.

$$\text{tauto} :: \Pi P : \text{Prop}.\text{option } \text{LT}(P)$$

Prop	logical sort for propositions
$\text{LT}(\cdot)$	lift logical term into computational-level types
	$\text{LT}(T) \triangleq \Sigma x : T.\text{unit}$

Example: tauto

A simple automated prover for propositional tautologies.
Given a proposition, attempt to construct a proof object for it.

$$\text{tauto} :: \Pi P : \text{Prop}.\text{option } \text{LT}(P)$$

Prop	logical sort for propositions
$\text{LT}(\cdot)$	lift logical term into computational-level types
	$\text{LT}(T) \triangleq \Sigma x : T.\text{unit}$

Example: tauto

A simple automated prover for propositional tautologies.
Given a proposition, attempt to construct a proof object for it.

$$\text{tauto} :: \Pi P : \text{Prop}.\text{option } \text{LT}(P)$$

Prop	logical sort for propositions
$\text{LT}(\cdot)$	lift logical term into computational-level types
	$\text{LT}(T) \triangleq \Sigma x : T.\text{unit}$

Example: tauto

A simple automated prover for propositional tautologies.
Given a proposition, attempt to construct a proof object for it.

$$\text{tauto} :: \Pi P : \text{Prop}.\text{option } \text{LT}(P)$$

Prop	logical sort for propositions
$\text{LT}(\cdot)$	lift logical term into computational-level types
	$\text{LT}(T) \triangleq \Sigma x : T.\text{unit}$

Example: tauto

$$
\begin{array}{llll}
\text{tauto } P & = & \text{holcase } P \text{ of} & \\
\quad P_1 \wedge P_2 & \mapsto & \text{do pf}_1 \leftarrow \text{tauto } P_1; & \\
 & & \quad \text{pf}_2 \leftarrow \text{tauto } P_2; & \\
 & & \quad \langle \cdots \textit{ proof of } P_1 \wedge P_2 \cdots \rangle & \\
\mid P_1 \vee P_2 & \mapsto & (\text{do pf}_1 \leftarrow \text{tauto } P_1; & \\
 & & \quad \langle \cdots \textit{ proof of } P_1 \vee P_2 \cdots \rangle) \mid\mid & \\
 & & (\text{do pf}_2 \leftarrow \text{tauto } P_2; & \\
 & & \quad \langle \cdots \textit{ proof of } P_1 \vee P_2 \cdots \rangle) & \\
\mid \text{True} & \mapsto & \text{Some } \langle \cdots \textit{ proof of } \text{ True} \cdots \rangle & \\
\mid P' & \mapsto & \text{None} &
\end{array}
$$

Handling binders

How to extend to handle quantification case?

$$
\begin{aligned}
&\mathsf{tauto}\ P \ = \ \mathsf{holcase}\ P\ \mathsf{of} \\
&\qquad \ldots \\
&\quad\ \mid \forall x : A.P' \quad \mapsto \quad \mathsf{do}\ \mathsf{pf} \leftarrow \mathsf{tauto}\ P'; \\
&\qquad\qquad\qquad\qquad\qquad\quad \langle \cdots \ \textit{proof of}\ \ \forall x : A.P' \ \cdots \rangle \\
&\quad\ \mid \ldots
\end{aligned}
$$

Handling binders

How to extend to handle quantification case?

$$
\begin{array}{l}
\mathsf{tauto}\ P \ = \ \mathsf{holcase}\ P\ \mathsf{of} \\
\quad \cdots \\
\quad \mid \forall x : A.P' \quad \mapsto \quad \mathsf{do}\ \mathsf{pf} \leftarrow \mathsf{tauto}\ P'; \\
\qquad\qquad\qquad\qquad\qquad \langle \cdots\ \textit{proof of}\ \forall x : A.P' \cdots \rangle \\
\quad \mid \cdots
\end{array}
$$

Need to track the fact that P' and pf refer to an extended context compared to P

Handling binders: contextual terms

Modeled after contextual modal type theory, as used in Beluga.
Use *contextual terms* – terms carrying the context they refer to

$$T ::= [\Phi]\, t \qquad \text{Typing: } \frac{\Phi \vdash t : t'}{\vdash [\Phi]\, t : [\Phi]\, t'}$$

Introduce *contextual variables* or *metavariables* – variables standing for contextual terms, and the associated environment $\mathcal{M}$

$$\mathcal{M} ::= \bullet \mid X : T$$

Computational language manipulates contextual terms instead of simple logical terms, therefore binds metavariables.

$$\tau ::= \cdots \mid \Pi X : T.\tau \mid \Sigma X : T.\tau$$

Handling binders: contextual terms

Need to extend the logic, so that we can use metavariables as part of logical terms.

$$t ::= \cdots \mid X/\sigma$$

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

X will eventually be substituted with a term with free variables coming from an environment Φ. The substitution σ needs to map such variables into terms well-typed under the current Φ' context.

Handling binders: parametric contexts

- Need to be able to write functions that work with terms living in any context
- Therefore need a notion of quantification over contexts in the computational language.
- Simple extension to contextual terms and variables as presented so far.

Handling binders: contextual terms

Type of tauto becomes:

$$\text{tauto} :: \Pi\Phi : \text{ctx}.\Pi P : [\Phi]\,\text{Prop}.\text{option LT}([\Phi]\,P)$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\text{tauto } \Phi \; P \;=\; \text{holcase } P \text{ of}} \\
\quad \ldots & & \\
\quad \mid \forall x : A.P' & \mapsto & \text{do pf} \leftarrow \text{tauto } (\Phi,\; x : A)\; P'; \\
& & \quad \langle [\Phi]\, \lambda x : A.\text{pf}/[\text{id}_\Phi,\; x] \rangle \\
\quad \mid P_1 \rightarrow P_2 & \mapsto & \text{do pf} \leftarrow \text{tauto } (\Phi,\; x : P_1)\; P_2; \\
& & \quad \langle [\Phi]\, \lambda x : P_1.\text{pf}/[\text{id}_\Phi,\; x] \rangle \\
\quad \mid \ldots & &
\end{array}
$$

Example: tauto

- Strong static guarantees (similar to proof-by-reflection)
- Easy to extend (e.g. recursively prove premises of hypotheses)

Example: Deciding equality

Example: deciding equality

- Given a list of proofs of equality between terms, decide whether two terms are equal.
- Common algorithms for this procedure use some form of an imperative union-find data structure in order to be efficient.

Simple algorithm for deciding equality

- Maintain a union-find data-structure that represents equivalence classes.
- Each equivalence class has one representative term.
- Each term we are interested in refers to a parent term that belongs to the same equivalence class.
- If a term refers to itself, it is considered to be the representative of its equivalence class.
- When a new equality between two terms is processed, find their representatives; if they are not the same, merge the two equivalence classes by making one representative the parent of the other.

Representing the union-find data-structure

- We choose to encode the union-find data structure as a hash table.
- Each term gets mapped into a value representing its parent term.
- We also want to yield proofs, so we store a proof object on how a term is equal to its parent.
- We have a built-in hash function for logical terms.

Representing the union-find data-structure

The type for the hash table will be:

$$\text{eqhash} = \text{array}\ (\Sigma X : T.\Sigma X' : T.\mathsf{LT}(X = X'))$$

Each array element is a key-value pair, mapping one term, the key, to its value: its parent term, and a proof witnessing that the two terms are equal.

Representing the union-find data-structure

The type for the hash table will be:

$$\text{eqhash} = \text{array}\ (\Sigma X : T.\Sigma X' : T.\mathsf{LT}(X = X'))$$

Each array element is a key-value pair, mapping one term, the key, to its value: its parent term, and a proof witnessing that the two terms are equal.

Representing the union-find data-structure

The type for the hash table will be:

$$\text{eqhash} = \text{array}\ (\Sigma X : T.\Sigma X' : T.\mathsf{LT}(X = X'))$$

Each array element is a key-value pair, mapping one term, the key, to its value: its parent term, and a proof witnessing that the two terms are equal.

Representing the union-find data-structure

The type for the hash table will be:

$$\text{eqhash} = \text{array}\ (\Sigma X : T.\Sigma X' : T.\mathsf{LT}(X = X'))$$

Each array element is a key-value pair, mapping one term, the key, to its value: its parent term, and a proof witnessing that the two terms are equal.

Representing the union-find data-structure

The type for the hash table will be:

$$\text{eqhash} = \text{array}\ (\Sigma X : T.\Sigma X' : T.\mathsf{LT}(X = X'))$$

Each array element is a key-value pair, mapping one term, the key, to its value: its parent term, and a proof witnessing that the two terms are equal.

Types for the main functions

find:

given a term and a hash-table, return the representative of its equivalence class, plus a proof of equivalence

$$\Pi X : T.\mathsf{eqhash} \rightarrow \Sigma X' : T.\mathsf{LT}(X = X')$$

Types for the main functions

union:

given two terms, and a proof of their equivalence, update the hash-table accordingly

$$\Pi X : T.\Pi X' : T.\Pi \mathsf{pf} : X = X'.\mathsf{eqhash} \to \mathsf{unit}$$

Types for the main functions

areEqual?:

given two terms and a hash-table, determine whether they're equal or not

$$\Pi X : T.\Pi X' : T.\mathsf{eqhash} \rightarrow \mathsf{option}\ \mathsf{LT}(X = X')$$

Deciding equality

- Simple to adapt the involved data structures in VeriML
- While also yielding proofs
- Termination non-trivial, but didn't need to prove it

Metatheory & implementation

Metatheory

- Developed the type system and small-step operational semantics for VeriML
- Proved progress and preservation for the language
- Proof for normal ML features orthogonal to logic-related features
- Details in our upcoming ICFP 2010 paper, plus accompanying TR

Metatheory

- If we disallow pattern-matching over proof objects, we can prove that semantics are preserved even if proof objects are erased
- Type-safety guarantees that valid proof objects exist in principle
- Depending on wanted level of assurance, we can choose to produce such proof objects or not

Implementation

- Prototype implementation in OCaml
- About 5k lines of code
- Implementation of logic is about 800 lines (trusted base)
- Examples that type-check and run:
  - Propositional tautologies prover
  - Extension to also use equalities with uninterpreted functions
  - Conversion of formulas to NNF

Directions and future work

Proof assistants

- Use the language in order to develop the infrastructure of a proof assistant
- Type-safe proof scripts that don't need to generate proof objects

Questions:

- How to represent proof states
- How to code the basic LCF tactics
- How should interactivity be handled

Directions and future work

Dependently typed languages

- Programming in languages like Agda, Epigram, etc. involves a form of theorem proving
- Evidenced in the Russell framework
- Can we provide a way to automate part of this using a similar computational language?
- Pattern matching is form of typecase construct

Directions and future work

Certifying compilers, static analysis tools

- Leverage the language to write such tools that produce proofs
- Use proof object erasure to avoid runtime costs

Thanks a lot!

More info:

http://flint.cs.yale.edu/publications/veriml.html

http://zoo.cs.yale.edu/~ams257