# An Operational Foundation for the Tactic Language of Coq

Wojciech Jedynak
Institute of Computer Science
University of Wrocław
wjedynak@gmail.com

Małgorzata Biernacka
Institute of Computer Science
University of Wrocław
mabi@cs.uni.wroc.pl

Dariusz Biernacki
Institute of Computer Science
University of Wrocław
dabi@cs.uni.wroc.pl

## ABSTRACT

We introduce a semantic toolbox for Ltac, the tactic language of the popular Coq proof assistant. We present three formats of operational semantics, each of which has its use in the practice of tactic programming: a big-step specification in the form of natural semantics, a model of implementation in the form of an abstract machine, and a small-step characterization of computation in the form of reduction semantics. The three semantics are provably equivalent and have been obtained via off-the-shelf derivation techniques of the functional correspondence and the syntactic correspondence. We also give examples of Ltac programs and discuss some of the issues that the formal semantics help to clarify.

With this work we hope to enhance the operational understanding of Ltac as well as to set up a framework to reason about Coq scripts and to build tools supporting tactic programming based on rigorous semantics.

## Categories and Subject Descriptors

D.3.1 [**PROGRAMMING LANGUAGES**]: Programming Languages—Formal definitions and theory; F.3.2 [**LOGICS AND MEANINGS OF PROGRAMS**]: Semantics of Programming Languages—Operational semantics; F.4.1 [**MATHEMATICAL LOGIC AND FORMAL LANGUAGES**]: Mathematical Logic—Mechanical theorem proving

## General Terms

Languages, Theory

## Keywords

proof assistant Coq, tactics, natural semantics, abstract machine, reduction semantics

## 1. INTRODUCTION

After several decades of work on proof assistants and logical frameworks the state-of-the-art tools have become pretty mature and widely used. Many reported successes show that it is now possible to carry out large-scale realistic developments in computer science and to provide machine-checkable proofs for whole areas of mathematics. Some representative projects in the former category include a certified compiler for a C-like imperative language (the CompCert project [25]) in Coq [3], the verification of a whole microkernel (the seL4 project [20]) in Isabelle [29] or the formalization of the type safety of Standard ML [23] in Twelf [32]. The latter category includes the work on the Four Colour Theorem [14] and the Odd Order Theorem [15] led by Gonthier and done in Coq.

In some areas of computer science the use of proof assistants is particularly beneficial: they not only help ensure correctness but some of them also allow a high degree of proof automation, thus relieving the burden of formal verification and supporting the process of finding a proof.

The more widespread the use of proof assistants, the more pressing the need for tools to support proof development and refactorization. In [6], Bourke et al. describe their experiences of participating in some of the projects mentioned above and analyze the challenges of large-scale proof management.

Most big verification and formalization projects use a procedural tactic-based approach to theorem proving. In this approach the user constructs the proof indirectly by using tactics (if the system constructs an explicit proof term, it is hidden from the user). User-defined tacticals can be used to increase automation, to improve maintainability and reliability of proof scripts. In particular, well-designed generic tactics can drastically remove repetition and make the script resilient to small representation changes.

Coq is one of the popular proof assistants endowed with a powerful tactic language Ltac due to Delahaye. Unfortunately, the semantics of Ltac has been presented only informally, both by the author [11, 12] and in the documentation, including the reference manual for the latest release [35]. While the provided informal description is good enough as an introduction to the system, many of the features and corner cases are left unspecified or imprecise. Expert users take advantage of those unspecified behaviors, also to good use – we will discuss some examples of useful tactics in Section 2.3. These examples are taken from a Coq textbook, yet inexperienced users have to rely on intuition and experimentation to understand the mechanisms used to structure them.

As far as we are aware, the only effort to formally capture the semantics of Ltac has been done by Kirchner [18] who provided a small-step operational semantics in the form of


*PPDP '13,* September 16 - 18 2013, Madrid, Spain

http://dx.doi.org/10.1145/2505879.2505890.

a reduction semantics. However, he considered Ltac as originally described by Delahaye, so his work pertains to an obsolete version of Ltac. Moreover, he made some simplifications in the treatment of the exception mechanism of Ltac and thus does not give it full justice.

In this work we attempt to fill the voids in the formal treatment of Ltac described above by performing a comprehensive semantic study of the Coq tactic language and by providing a coherent toolset of formal semantics of different flavors. To this end, we propose an operational foundation for Ltac by means of:

- a natural semantics that is well suited to both quickly grasp the meaning of various constructs by the user as well as a basis for studying equivalence of tactics;
- an abstract machine that provides a model of implementation;
- a reduction semantics that describes the small-step computation and can be useful for execution tracing.

The starting point of this work is the natural semantics for a core subset of Ltac (dubbed CoreLtac) from which we derive the other two formats by adapting existing techniques of the functional correspondence [1] and the syntactic correspondence [9].

We postulate that these semantic artifacts can form a basis both for the formal study of Ltac and for the design of its variants or extensions. Hopefully, it might also serve as a reference point for its users.

The rest of the article is organized as follows: In Section 2 we review the general idea of tactics and discuss practical examples. We next motivate the need for a formal semantics by analyzing useful but possibly confusing examples of Ltac scripts. In Section 3 we introduce CoreLtac, a representative subset of Ltac. In Section 4 we present a natural semantics for CoreLtac. In Section 5 we derive the corresponding abstract machine. In Section 6 we extract a reduction semantics from the abstract machine. In Section 7 we discuss some possible variants and extensions of our development. In Section 8 we survey related work and in Section 9 we conclude and sketch directions for future work.

# 2. TACTICS, TACTICALS AND LTAC

In this section we provide a short discussion of some of the features of Ltac.

## 2.1 Tactic-based proving

Coq is a proof assistant that takes advantage of the proofs-as-programs paradigm (or the so-called Curry-Howard isomorphism) in that proofs of theorems are identified with terms of the corresponding type. The consequence of this fact is that the user can prove a theorem simply by writing the entire proof term directly and explicitly.

A more convenient approach, employed by some proof assistants (e.g. Coq, PVS, HOL, Isabelle) is to work interactively by using *tactics*, i.e., procedures that allow backward reasoning and that construct (parts of) the proof terms upon execution.

Basic building blocks in tactic-based programming are atomic tactics that roughly correspond to single inference rules of the logic underlying the system. This level of granularity also quickly becomes impractical – bigger proofs become bloated and unmaintainable. As one of the means for proof structuring, Coq provides *tacticals* (tactic combinators, higher-order tactics).

Many available tacticals date back all the way to LCF and are more than 30 years old. The most fundamental combinator is the sequential composition denoted $t_1; t_2$. Its informal meaning is "apply $t_2$ to every subgoal produced by the execution of $t_1$ in the current proof context."

Whenever a tactic cannot be applied successfully, the execution results in a failure which is reported back to the user who can then try some other tactic. This allows for a unique experience: at each step the user can blindly invoke a set of her favorite tactics, hoping that the goal is simple enough for the machine to resolve. Only otherwise need the proof context be manually introspected.

Coq's tacticals are very useful for combining existing decision procedures into a single tactic. The failure-based notion of control allows one to create powerful heuristics. For instance, first $[t_1|t_2|...|t_n]$ executes the tactics $t_1, t_2, \ldots t_n$ one by one until a non-failing one is found; otherwise the whole first fails. The following example presents a typical use case:

```
induction e;
  first [ assumption | trivial | congruence
        | intuition | auto | eauto with arith ].
```

The execution of the induction tactic can generate dozens of cases that may vary in complexity but often many of them are easy enough to be solved automatically. However, some automations can be time expensive, so it makes sense to try the cheap ones first.

Coq comes equipped with more than a hundred built-in tactics of varying range of generality and granularity, including sophisticated decision procedures for arithmetic theories, propositional and first-order logic and others. Some of them are very general and take long to complete, so in some cases the user has to instrument the system to use them only under certain conditions. Ltac's matching constructions fit this role nicely.

## 2.2 Practical examples of Ltac scripts

In this section we show example tactics that demostrate typical use cases of most of Ltac's distinctive features. An interested reader can find many examples of sophisticated tactics in Chlipala's textbook [7].

### 2.2.1 Assumption reimplementation

Pattern backtracking allows us to provide a concise implementation of the built-in assumption tactic:

```
Ltac my_assumption :=
  match goal with
  | [ H : _ |- _ ] => exact H
  end.
```

To analyze this example, we must review the semantics of Coq's matching constructions. match goal provides pattern backtracking upon failure: all possible matchings of the goal against the given pattern are tried out in turn: if the tactic on the right hand side of the clause fails, *then it is backtracked*, and another instantiation (for the same clause) is tried.

[ H : _ |- _ ] is a proof context pattern that can match any assumption and exact fails when the goal is not immediate from the argument. Here, because of the backtracking semantics of match goal, exact will be applied to every as-

sumption until the goal is solved or no assumption matches.

### 2.2.2 A decision procedure for the theory of booleans

Next we present a simple decision procedure for the theory of booleans.

```
Ltac analyze_bool_cases :=
  repeat
    match goal with
    | [ H : bool |- _ ] => destruct H
    end;
  simpl in *; congruence.
```

Since bool has only two inhabitants, we can perform case analysis on all booleans in the context and hope that the goal can be proved using simplification and equational reasoning. Proof context patterns can pattern-match inside the type of the goal and assumptions, and we use this possibility to filter boolean assumptions; repeat makes sure that we destruct all booleans.

### 2.2.3 Contradiction by asymmetry

We demonstrate the use of non-linear pattern matching for automatic lemma argument instantiation.

```
Axiom asymmetry: forall n m,
  n < m -> m < n -> False.

Ltac contradiction_by_asymmetry :=
  match goal with
  | [ H1 : ?n < ?m, H2 : ?m < ?n |- _ ] =>
    elim (asymmetry H1 H2)
  end.
```

To use the asymmetry lemma, we need to find a suitable pair of arguments $n, m$. It is possible to perform this instantiation by hand (by hardwiring the names from the proof context), but that would make the script very brittle – if the names change in the future, the script will break. An automated tactic has the advantage that it can be easily reused, making the script more compositional.

### 2.2.4 Term normalization example

Here we demonstrate the use of let, eval and change for term simplification.

```
Ltac my_cbv x :=
  let y := eval cbv in x in
  change x with y
```

We bind $y$ to the normalized version of $x$ and then use change to replace all occurrences of $x$ with $y$.

## 2.3 Complex tactics and pitfalls

Consider the following Ltac tactic, taken from the Coq textbook by Chlipala [7]:

```
Ltac notHyp P :=
  match goal with
  | [ _ : P |- _ ] => fail 1
  | _ =>
    match P with
    | ?P1 /\ ?P2 =>
        first [ notHyp P1 | notHyp P2 | fail 2 ]
    | _ =>
        idtac
    end
 end.
```

The intention is that notHyp should succeed whenever there exists a subformula of $P$ that is not among the assumptions of the proof context. Chlipala uses notHyp to prevent his decision procedure for propositional logic from extending the proof context with the same proposition over and over again. To comprehend this example it is crucial to understand the interplay between failure levels, first and match goal's clause and pattern backtracking.

In the documentation of the system the description of most tacticals includes the behavior in case of argument failure, but it is only true for failure at level 0. To analyze this tactic, we need to know what happens when the active argument of first yields an exception with a positive level. Chlipala's explanation suggests (and our experiments confirm it) that in that case the level is decremented and the exception is rethrown.

In his book Chlipala gives numerous examples like this one that uncover and take advantage of various subtleties of Ltac; these examples are not artificial but quite useful. By providing rigorous account of Ltac semantics, we aim to facilitate analysis and reasoning about Coq proof scripts, especially when advanced features are combined (perhaps abused) in non-obvious and creative ways.

# 3. CORE LTAC

In this section we present the syntax of CoreLtac, a subset of Ltac that is representative of the full language and rich enough to illustrate many of the most interesting features of Ltac. In the full formalization we plan to include more features common in functional languages, such as recursion or pattern matching on terms as well as to scale the development to cover all of Ltac.

It should be noted that CoreLtac is designed for presentational purposes and not intended as an intermediate tactic language. We therefore include multi-argument functions as present in Ltac in contrast with the use of currying in the lambda calculus.

We introduce the following syntactic categories:

$$
\begin{array}{rrcl}
\text{(expression)} & e & ::= & v \\
 & & \mid & x \\
 & & \mid & e\ es \\
 & & \mid & \mathsf{let}\ x := e_1\ \mathsf{in}\ e_2 \\
 & & \mid & \mathsf{mgoal}\ \vec{cl} \\
\text{(expression list)} & es & ::= & \varepsilon \mid (e : es) \\
\text{(value)} & v & ::= & t \mid \lceil n \rceil \mid \lambda \vec{x}.e \\
\text{(tactic)} & t & ::= & \mathsf{idtac} \\
 & & \mid & \mathsf{fail}\ a \\
 & & \mid & e_1; e_2 \\
 & & \mid & \mathsf{repeat}\ e \\
 & & \mid & \mathsf{first}\ es \\
 & & \mid & \mathsf{progress}\ e \\
\text{(atom)} & a & ::= & x \mid \lceil n \rceil \\
\text{(clause)} & cl & ::= & (p, e)
\end{array}
$$

The grammar of expressions includes, apart from standard functional constructs, a match-goal expression that performs a form of pattern matching on goal. We note that an application is in a general form: an expression can be applied to a list of expressions as arguments. In Ltac, the user is more constrained in that in the function position one can only use an identifier, but we relax this requirement for technical rea-

sons. Values include tactics, natural numbers, and lambda abstractions.

The tactic language is built on top of a logic with its proof terms. In case of Coq the logic is the Calculus of Inductive Constructions (CIC), and Gallina is its specification language. The grammar of Gallina terms and the inference rules of CIC are largely orthogonal to our development, therefore we do not give a full account of them here. For the same reason, we do not explicitly introduce the atomic tactics of Coq.

Similarly, we do not commit to any particular grammar of proof context patterns. The natural semantics, presented in the next section, treats proof context matching in a modular fashion by introducing an abstract representation of instantiation candidate generation.

# 4. NATURAL SEMANTICS

Natural semantics is a big-step operational semantics that defines the relation between a term and the result of its evaluation in terms of the meaning of its subterms. It is particularly convenient for formalizing meaning of programs in a high-level, human-readable form and to reason about equivalence of programs, correctness of program transformations, etc.

In this section we introduce natural semantics of CoreLtac. The semantics faithfully accounts for Ltac behavior for the constructs of the core language. It is based on the informal semantics of Ltac for Coq v.8.4 presented in the manual as well as on hands-on experiments with the system, especially to resolve corner cases [35].

CoreLtac is a language that combines pure functional behavior of expressions that can be evaluated with imperative constructs that modify the state, i.e., the current goal. A goal in Coq's terminology is simply a logical judgment that we try to prove by executing a tactic. The execution of a tactic can change the goal into a new goal or a list of goals. The proof is finished when the tactic produces the empty list of goals. Apart from being successful, a tactic can raise an error that in Coq signals roughly two types of failure: a dynamic type error (CoreLtac is untyped), or a "wrong tactic" failure which means that the current tactic cannot prove or simplify the current goal. The latter failure can be also prescribed by the programmer by means of the fail tactic which combined with the backtracking semantics of match-goal gives the user a powerful tool for automating proof search.

The informal semantics indicates that in CoreLtac a tactic can be seen as a special kind of expression that can be evaluated and then executed. A Coq proof script consists of a sequence of tactics, each followed by a dot indicating the execution of the tactic. Therefore, in the following, a statement of the form $t.$ denotes a tactic to be executed (as in a Coq script).

The natural semantics is shown in Figures 2 and 3. It uses two main judgments reflecting the natural distinction between evaluation and execution. In the following, we explain the two modes of operation and comment on some of the rules.

The $\oplus$ operator denotes concatenation of goal lists, used to flatten lists of subgoals into a single subgoal list. $[G]$ denotes the singleton list containing $G$.

## 4.1 The notion of modes

Consider the following (artificial) Ltac script:

```
let x := idtac in first [ auto | x ].
```

While idtac and first are both tacticals, it is clear that we want $x$ to be bound not to the result of the *execution* of idtac, but to the idtac tactic itself. On the other hand, we do want to actually execute the first tactic.

Generalizing, we come to the realization that given the script:

```
let x := e1 in e2.
```

we should treat $e_1$ and $e_2$ differently: while both $e_1$ and $e_2$ should be *evaluated*, only $e_2$ should be *executed* in the next step.

To make this distinction precise, we say that a computation can operate in one of two *modes*: $v$ for evaluation and $x$ for execution. As a consequence, in the natural semantics we have two main judgments, one for each mode.

## 4.2 The judgments

The first judgment denotes *tactic execution* and is written

$$G \triangleright t \downarrow_x r_x$$

The result can be either a list of subgoals $Gs$ or a level-$n$ failure $\bot_n$, the semantic representation of fail $\lceil n \rceil$. We say that a tactic *solves* the goal $G$ when $Gs$ is the empty list.

The second judgment formalizes *expression evaluation* and is written

$$G \triangleright e \downarrow_v r_v$$

The result can be a value $v$, a list of subgoals $Gs$ or a level-$n$ failure $\bot_n$. It is somewhat surprising that we include $Gs$ in $r_v$. This is caused solely by the distinct interaction between let and match goal. Otherwise the reader is free to ignore this possibility in the context of almost any other rule.

In our presentation expression evaluation and tactic execution are composed in one of two ways. First of all, we need to be able to *execute an expression*. This will be written

$$G \triangleright e \Downarrow r_x$$

This judgment can be summarized as

1. Evaluate $e$ to tactic $t$
2. Execute $t$

Formal rules (given in Figure 1, top row) also state that if evaluation does not yield a tactic then the whole computation fails. In a typical programming language, a situation like this would be called a type error and could lead to a stuck term. In Ltac it is however possible to trap (and recover from) virtually all errors, because they get reported as failures.

Throughout the whole interaction with the user, the proof engine maintains a stack of open subgoals (the goal stack). Initially it contains a single element – the theorem the user wants to prove. When the goal stack becomes empty, the original goal is solved and the theorem is proved. Whenever a top-level tactic execution is requested (in Coq, this is the dot command, denoted $e.$), the tactic e is applied on the top of the stack and new subgoals thus generated are pushed

$$\textsf{EEX}_1\ \frac{G \rhd e \downarrow_v Gs}{G \rhd e \Downarrow Gs} \qquad \textsf{EEX}_2\ \frac{\begin{array}{c}G \rhd e \downarrow_v t\\ G \rhd t \downarrow_x r_x\end{array}}{G \rhd e \Downarrow r_x} \qquad \textsf{EEX}_3\ \frac{G \rhd e \downarrow_v \bot_n}{G \rhd e \Downarrow \bot_n} \qquad \textsf{EEX}_4\ \frac{\begin{array}{c}G \rhd e \downarrow_v v\\ v = \lceil n \rceil \mid \lambda\vec{x}.e\end{array}}{G \rhd e \Downarrow \bot_0}$$

$$\textsf{EEV}_1\ \frac{G \rhd e \downarrow_v Gs}{G \rhd e \downarrow_{vx} Gs} \qquad \textsf{EEV}_2\ \frac{\begin{array}{c}G \rhd e \downarrow_v t\\ G \rhd t \downarrow_x r_x\end{array}}{G \rhd e \downarrow_{vx} r_x} \qquad \textsf{EEV}_3\ \frac{G \rhd e \downarrow_v \bot_n}{G \rhd e \downarrow_{vx} \bot_n} \qquad \textsf{EEV}_4\ \frac{\begin{array}{c}G \rhd e \downarrow_v v\\ v = \lceil n \rceil \mid \lambda\vec{x}.e\end{array}}{G \rhd e \downarrow_{vx} v}$$

**Figure 1:** Natural semantics – expression execution (top) and extended expression evaluation (bottom)

$$\textsf{IDTAC}\ \frac{}{G \rhd \textsf{idtac} \downarrow_x [G]} \qquad \textsf{FAIL}\ \frac{}{G \rhd \textsf{fail}\ \lceil n \rceil \downarrow_x \bot_n} \qquad \textsf{FIRST}_1\ \frac{}{G \rhd \textsf{first}\ \varepsilon \downarrow_x \bot_0}$$

$$\textsf{FIRST}_2\ \frac{G \rhd e \Downarrow Gs}{G \rhd \textsf{first}\ (e : es) \downarrow_x Gs} \qquad \textsf{FIRST}_3\ \frac{G \rhd e \Downarrow \bot_{\textsf{s}(n)}}{G \rhd \textsf{first}\ (e : es) \downarrow_x \bot_n} \qquad \textsf{FIRST}_4\ \frac{G \rhd e \Downarrow \bot_0 \qquad G \rhd \textsf{first}\ es \downarrow_x r_x}{G \rhd \textsf{first}\ (e : es) \downarrow_x r_x}$$

$$\textsf{PROGR}_1\ \frac{G \rhd e \Downarrow \bot_n}{G \rhd \textsf{progress}\ e \downarrow_x \bot_n} \qquad \textsf{PROGR}_2\ \frac{G \rhd e \Downarrow [G]}{G \rhd \textsf{progress}\ e \downarrow_x \bot_0} \qquad \textsf{PROGR}_3\ \frac{G \rhd e \Downarrow Gs \qquad Gs \neq [G]}{G \rhd \textsf{progress}\ e \downarrow_x Gs}$$

$$\textsf{SEMI}_1\ \frac{G \rhd e_1 \Downarrow \bot_n}{G \rhd e_1; e_2 \downarrow_x \bot_n} \qquad \textsf{SEMI}_2\ \frac{G \rhd e_1 \Downarrow Gs \qquad Gs \rhd e_2 \downarrow_{seq} r_x}{G \rhd e_1; e_2 \downarrow_x r_x}$$

$$\textsf{REP}_1\ \frac{G \rhd \textsf{progress}\ e \downarrow_x \bot_0}{G \rhd \textsf{repeat}\ e \downarrow_x [G]} \qquad \textsf{REP}_2\ \frac{G \rhd \textsf{progress}\ e \downarrow_x \bot_{\textsf{s}(n)}}{G \rhd \textsf{repeat}\ e \downarrow_x \bot_n} \qquad \textsf{REP}_3\ \frac{\begin{array}{c}G \rhd \textsf{progress}\ e \downarrow_x Gs\\ Gs \rhd (\textsf{repeat}\ e) \downarrow_{seq} r_x\end{array}}{G \rhd \textsf{repeat}\ e \downarrow_x r_x}$$

$$\textsf{SEQ}_1\ \frac{}{\varepsilon \rhd e \downarrow_{seq} \varepsilon} \qquad \textsf{SEQ}_2\ \frac{G \rhd e \Downarrow \bot_n}{(G : Gs) \rhd e \downarrow_{seq} \bot_n} \qquad \textsf{SEQ}_3\ \frac{\begin{array}{c}G \rhd e \Downarrow Gs'\\ Gs \rhd e \downarrow_{seq} \bot_n\end{array}}{(G : Gs) \rhd e \downarrow_{seq} \bot_n} \qquad \textsf{SEQ}_4\ \frac{\begin{array}{c}G \rhd e \Downarrow Gs'\\ Gs \rhd e \downarrow_{seq} Gs''\end{array}}{(G : Gs) \rhd e \downarrow_{seq} Gs' \oplus Gs''}$$

**Figure 2:** Natural semantics – tactic execution

$$\textsf{VAL}\ \frac{}{G \rhd v \downarrow_v v} \qquad \textsf{APP}_1\ \frac{e = \lceil n \rceil \mid t}{G \rhd e\ es \downarrow_v \bot_0} \qquad \textsf{APP}_2\ \frac{G \rhd es \downarrow_{args} \bot_n}{G \rhd (\lambda\vec{x}.e)\ es \downarrow_v \bot_n}$$

$$\textsf{APP}_3\ \frac{\begin{array}{c}G \rhd es \downarrow_{args} \vec{v}\\ |\vec{v}| < |\vec{x}|\\ (\vec{x_1}, \vec{x_2}) = \textsf{split}\ \vec{x}\ \textsf{at}\ |\vec{v}|\end{array}}{G \rhd (\lambda\vec{x}.e)\ es \downarrow_v \lambda\vec{x_2}.e[\overrightarrow{x_1} := \vec{v}]} \qquad \textsf{APP}_4\ \frac{\begin{array}{c}G \rhd es \downarrow_{args} \vec{v}\\ |\vec{v}| = |\vec{x}|\\ G \rhd e[\vec{x} := es] \downarrow_v r_v\end{array}}{G \rhd (\lambda\vec{x}.e)\ es \downarrow_v r_v} \qquad \textsf{APP}_5\ \frac{\begin{array}{c}G \rhd es \downarrow_{args} \vec{v}\\ |\vec{x}| < |\vec{v}|\\ (\vec{v}_1, \vec{v}_2) = \textsf{split}\ \vec{v}\ \textsf{at}\ |\vec{x}|\\ G \rhd e[\vec{x} := \vec{v}_1]\ \vec{v}_2 \downarrow_v r_v\end{array}}{G \rhd (\lambda\vec{x}.e)\ es \downarrow_v r_v}$$

$$\textsf{ARGS}_1\ \frac{}{G \rhd \varepsilon \downarrow_{args} \varepsilon} \qquad \textsf{ARGS}_2\ \frac{G \rhd e \Downarrow \bot_n}{G \rhd (e : es) \downarrow_{args} \bot_n} \qquad \textsf{ARGS}_3\ \frac{G \rhd e \Downarrow Gs}{G \rhd (e : es) \downarrow_{args} \bot_0}$$

$$\textsf{ARGS}_4\ \frac{G \rhd e \Downarrow v \qquad G \rhd es \downarrow_{args} \bot_n}{G \rhd (e : es) \downarrow_{args} \bot_n} \qquad \textsf{ARGS}_5\ \frac{G \rhd e \Downarrow v \qquad G \rhd es \downarrow_{args} \vec{v}}{G \rhd (e : es) \downarrow_{args} (v : \vec{v})}$$

$$\textsf{LET}_1\ \frac{G \rhd e_1 \downarrow_v \bot_n}{G \rhd \textsf{let}\ x := e_1\ \textsf{in}\ e_2 \downarrow_v \bot_n} \qquad \textsf{LET}_2\ \frac{G \rhd e_1 \downarrow_v Gs}{G \rhd \textsf{let}\ x := e_1\ \textsf{in}\ e_2 \downarrow_v \bot_0} \qquad \textsf{LET}_3\ \frac{G \rhd e_1 \downarrow_v v \qquad G \rhd e_2[x := v] \downarrow_v r_v}{G \rhd \textsf{let}\ x := e_1\ \textsf{in}\ e_2 \downarrow_v r_v}$$

$$\textsf{MG}_1\ \frac{}{G \rhd \textsf{mgoal}\ \varepsilon \downarrow_v \bot_0} \qquad \textsf{MG}_2\ \frac{G \rhd (\textsf{start}\ G\ p) \cdot e \cdot \vec{cl} \downarrow_{pattern} r_x}{G \rhd \textsf{mgoal}\ ((p, e) : \vec{cl}) \downarrow_v r_x} \qquad \textsf{PAT}_1\ \frac{\textsf{next}\ m = \textsf{Done} \qquad G \rhd \textsf{mgoal}\ \vec{cl} \downarrow_v r_x}{G \rhd m \cdot e \cdot \vec{cl} \downarrow_{pattern} r_x}$$

$$\textsf{PAT}_2\ \frac{\begin{array}{c}\textsf{next}\ m = \textsf{Match}(\sigma, m')\\ G \rhd \sigma(e) \downarrow_{vx} r_x\\ r_x = v \mid Gs\end{array}}{G \rhd m \cdot e \cdot \vec{cl} \downarrow_{pattern} r_x} \qquad \textsf{PAT}_3\ \frac{\begin{array}{c}\textsf{next}\ m = \textsf{Match}(\sigma, m')\\ G \rhd \sigma(e) \downarrow_{vx} \bot_{\textsf{s}(n)}\end{array}}{G \rhd m \cdot e \cdot \vec{cl} \downarrow_{pattern} \bot_n} \qquad \textsf{PAT}_4\ \frac{\begin{array}{c}\textsf{next}\ m = \textsf{Match}(\sigma, m')\\ G \rhd \sigma(e) \downarrow_{vx} \bot_0\\ G \rhd m' \cdot e \cdot \vec{cl} \downarrow_{pattern} r_x\end{array}}{G \rhd m \cdot e \cdot \vec{cl} \downarrow_{pattern} r_x}$$

**Figure 3:** Natural semantics – expression evaluation

back onto the stack. Given the dot command, Coq computes the response according to the expression evaluation relation.

To describe the computation of match goal clauses we need to alter the above behavior; the second composition is called *extended expression evaluation* and written

$$G \triangleright e \downarrow_{vx} r_{eeval}$$

The rules are presented in Figure 1 (bottom row). The only difference is that if evaluation yields a value $v$ that is not a tactic, then we do not consider it as an error and $v$ is the result. Since tactics are executed as before, the result can be a non-tactic value, a list of subgoals $Gs$ or a level-$n$ failure $\bot_n$.

Finally, we have helper judgments for sequential tactic execution, sequential argument evaluation and for pattern instantiation backtracking.

## 4.3 Tactic execution

The rules for tactic execution are shown in Figure 2.

progress $e$ performs expression execution on $e$ and, in case of a success, checks if the computation has been meaningful, i.e., the goal has been changed. repeat $e$ executes $e$ and then executes itself in each subgoal recursively, stopping upon failure. repeat wraps $e$ with progress to prevent infinite loops.[1]

Whenever a tactic expression $e$ needs to be applied in more than one goal, it makes use of a sequentialization mechanism that in the natural semantics is expressed with the judgment $Gs \triangleright e \downarrow_{seq} r$.

## 4.4 Expression evaluation

The rules for expression evaluation are shown in Figure 3.

Multiple argument application is handled by the judgment $G \triangleright es \downarrow_{args} r$.

To model pattern instantiation backtracking, we have the following judgment:

$$G \triangleright m \cdot t \cdot \vec{cl} \downarrow_{pattern} r_x$$

where $m$ is an abstract representation of the pattern matching functionality. Ltac has the unique behavior concerning pattern matching: if more than one hypothesis matches a given pattern then they are tried one-by-one until either evaluation of the right-hand-side of the clause (with the current substitution) succeeds or we run out of possible instantiations.

To reflect this behavior in the semantics, we assume two functions: start and next. Given a goal context $G$ and a pattern $p$, start initializes the pattern matching machinery. Given a $m$, next returns either Done when all possible instantiations have been tested, or Match$(\sigma, m')$. In that case, $\sigma$ is a functional representation of the substitution induced by the matching and $m'$ is the abstract object with updated internal state. The simplest and most natural implementation of the $m$ abstraction uses lazy lists – next is then a plain list destructor. Using the LogicT library introduced by Kiselyov et al. [19] it is also possible to implement this interface using continuations.

## 4.5 The let and match goal pitfall

match goal is a unique expression, because its evaluation may require tactic execution. This occurs precisely when the right hand side of a matching clause happens to evaluate to a tactic – it is then immediately executed. Therefore, the matching construction can return a list of subgoals.

On the other hand, to evaluate let $x := e_1$ in $e_2$ we first evaluate $e_1$ to a result $r$. $r$ is supposed to be a value, so when $r = \bot_n$ or $r = Gs$ the whole expression fails.

In practice, when we execute the script

```
let x := match goal with
          | _ => eauto
         end
in idtac x.
```

Coq produces the following error message[2]:

```
Error: Immediate match producing tactics not allowed in
local definitions.
```

When combining let with match goal as above, the user might intend one of the following:

1. To bind $x$ to the whole match goal expression and e.g. pass it as a parameter to a higher-order tactic (thus performing the matching later).

2. To bind $x$ to the right hand side of the clause matching right now.

In the first case the trick [7] is to coerce the match goal *expression* into a *tactic* by replacing it with

```
(idtac ; match goal with ... end)
```

In the second case one could think of delaying the computation using Ltac's lambdas, but recent versions of Coq provide a much cleaner solution – the lazymatch goal variant of match goal, which does not perform backtracking [35].

# 5. ABSTRACT MACHINE

In this section our goal is to obtain an abstract machine for CoreLtac. Traditionally, abstract machines were designed by hand, often in an ad-hoc manner [22, 21, 24, 31], which required skill and experience. In contrast, techniques developed by Danvy et al. [1, 9] allow one to obtain abstract machines mechanically, by performing transformations on existing semantics. In this section we take advantage of the technique know as *functional correspondence* [1].

The functional correspondence (in the original formulation by Ager et al. [1]) begins with an evaluator, e.g., implementing a natural semantics of a programming language, and consists of a conversion to CPS (continuation-passing style) followed by Reynolds's defunctionalization [34] giving as a result an abstract machine. Recently, Piróg and Biernacki have demonstrated in [33] that one can just as well begin with a natural semantics and perform the transformations not on programs, but on the rules of the semantics.

The abstract machine that we present in this section has been mechanically derived from the natural semantics of Section 4 using the functional correspondence. Transforming the natural semantics into defunctionalized continuation-passing style leads to the following mutually inductively defined grammars of stacks (that represent defunctionalized

[1]This was not the case in older versions of Coq.

[2]This may take a moment, as eauto can lead to a long proof search.

$$
\begin{array}{lcl}
\langle G, v, S_v\rangle^v_e & \Rightarrow & \langle S_v, v\rangle^v_a \\
\langle G, e\ es, S_v\rangle^v_e & \Rightarrow & \langle G, e, \mathsf{App}(G, es) : S_v\rangle^v_e \\
\langle G, \mathsf{let}\ x := e_1\ \mathsf{in}\ e_2, S_v\rangle^v_e & \Rightarrow & \langle G, e_1, \mathsf{Let}(G, x, e_2) : S_v\rangle^v_e \\
\langle G, \mathsf{mgoal}\ \varepsilon, S_v\rangle^v_e & \Rightarrow & \langle S_v, \bot_0\rangle^v_a \\
\langle G, \mathsf{mgoal}\ ((p, e) : \vec{cl}), S_v\rangle^v_e & \Rightarrow & \langle G, \mathsf{start}\ G\ p, e, \vec{cl}, S_v\rangle_p
\end{array}
$$

$$
\begin{array}{lcl}
\langle G, m, e, \vec{cl}, S_v\rangle_p & \Rightarrow & \langle G, \mathsf{mgoal}\ \vec{cl}, S_v\rangle^v_e \\
 & & \text{if next } m = \mathsf{Done} \\
\langle G, m, e, \vec{cl}, S_v\rangle_p & \Rightarrow & \langle G, \sigma(e), \mathsf{Pat}(G, e, m', \vec{cl}) : S_v\rangle^{vx}_e \\
 & & \text{if next } m = \mathsf{Match}(\sigma, m')
\end{array}
$$

$$
\begin{array}{lcl}
\langle G, \varepsilon, S_a\rangle^{as}_e & \Rightarrow & \langle S_a, \varepsilon\rangle^a_a \\
\langle G, e : es, S_a\rangle^{as}_e & \Rightarrow & \langle G, e, \mathsf{Args}(G, es) : S_a\rangle^v_e
\end{array}
$$

$$
\begin{array}{lcl}
\langle G, \mathsf{idtac}, S_x\rangle^x_e & \Rightarrow & \langle S_x, [G]\rangle^x_a \\
\langle G, \mathsf{fail}\ n, S_x\rangle^x_e & \Rightarrow & \langle S_x, \bot_n\rangle^x_a \\
\langle G, \mathsf{progress}\ e, S_x\rangle^x_e & \Rightarrow & \langle G, e, \mathsf{Prog}(G) : S_x\rangle^{ee}_e \\
\langle G, \mathsf{first}\ \varepsilon, S_x\rangle^x_e & \Rightarrow & \langle S_x, \bot_0\rangle^x_a \\
\langle G, \mathsf{first}\ (e : es), S_x\rangle^x_e & \Rightarrow & \langle G, e, \mathsf{First}(G, es) : S_x\rangle^{ee}_e \\
\langle G, e_1; e_2, S_x\rangle^x_e & \Rightarrow & \langle G, e_1, \mathsf{Semi}(e_2) : S_x\rangle^{ee}_e \\
\langle G, \mathsf{repeat}\ e, S_x\rangle^x_e & \Rightarrow & \langle G, \mathsf{progress}\ e, \mathsf{Rep}(G, e) : S_x\rangle^x_e
\end{array}
$$

$$
\begin{array}{lcl}
\langle \varepsilon, e, S_x\rangle_s & \Rightarrow & \langle S_x, \varepsilon\rangle^x_a \\
\langle G : Gs, e, S_x\rangle_s & \Rightarrow & \langle G, e, \mathsf{Seq}_1(Gs, e) : S_x\rangle^{ee}_e
\end{array}
$$

$$
\langle G, e, S_v\rangle^{vx}_e \Rightarrow \langle G, e, \mathsf{EExec}_1(G) : S_v\rangle^v_e
$$

$$
\langle G, e, S_x\rangle^{ee}_e \Rightarrow \langle G, e, \mathsf{EExpr}(G) : S_x\rangle^v_e
$$

**Figure 4:** Abstract machine – expression evaluation and tactic execution

$$
\begin{array}{lcl}
\langle \mathsf{Args}_1(G, \lambda\vec{x}.e) : S_v, \bot_n\rangle^a_a & \Rightarrow & \langle S_v, \bot_n\rangle^v_a \\
\langle \mathsf{Args}_1(G, \lambda\vec{x}.e) : S_v, \vec{v}\rangle^a_a & \Rightarrow & \begin{cases} |\vec{v}| < |\vec{x}| \rightarrow \underline{\mathsf{let}}\ \ (\vec{x_1}, \vec{x_2}) = \mathsf{split}\ \vec{x}\ \mathsf{at}\ |\vec{v}|\ \ \underline{\mathsf{in}}\ \langle S_v, \lambda\vec{x_2}.e[\overline{\vec{x_1} := \vec{v}}]\rangle^v_a \\ |\vec{v}| = |\vec{x}| \rightarrow \langle G, e[\overline{\vec{x} := \vec{v}}], S_v\rangle^v_e \\ |\vec{v}| > |\vec{x}| \rightarrow \underline{\mathsf{let}}\ \ (\vec{v_1}, \vec{v_2}) = \mathsf{split}\ \vec{v}\ \mathsf{at}\ |\vec{x}|\ \ \underline{\mathsf{in}}\ \langle G, e[\overline{\vec{x} := \vec{v_1}}]\ \vec{v_2}, S_v\rangle^v_e \end{cases} \\
\langle \mathsf{Args}_2(v) : S_a, \bot_n\rangle^a_a & \Rightarrow & \langle S_a, \bot_n\rangle^a_a \\
\langle \mathsf{Args}_2(v) : S_a, \vec{v}\rangle^a_a & \Rightarrow & \langle S_a, v : \vec{v}\rangle^a_a
\end{array}
$$

**Figure 5:** Abstract machine – function application

continuations):

$$
\begin{array}{rrcl}
\text{(evaluation stack)} & S_v & ::= & \mathsf{Let}(G, x, e) : S_v \\
 & & \mid & \mathsf{App}(G, es) : S_v \\
 & & \mid & \mathsf{Args}(G, es) : S_a \\
 & & \mid & \mathsf{Pat}(G, m, e, \vec{cl}) : S_v \\
 & & \mid & \mathsf{EExec}_1(G) : S_v \\
 & & \mid & \mathsf{EExpr}(G) : S_x \\
\text{(argument stack)} & S_a & ::= & \mathsf{Args}_1(G, \lambda\vec{x}.e) : S_v \\
 & & \mid & \mathsf{Args}_2(v) : S_a \\
\text{(execution stack)} & S_x & ::= & \mathsf{Nil} \\
 & & \mid & \mathsf{Prog}(G) : S_x \\
 & & \mid & \mathsf{First}(G, es) : S_x \\
 & & \mid & \mathsf{Semi}(e) : S_x \\
 & & \mid & \mathsf{Rep}(G, e) : S_x \\
 & & \mid & \mathsf{Seq}_1(Gs, e) : S_x \\
 & & \mid & \mathsf{Seq}_2(Gs) : S_x \\
 & & \mid & \mathsf{EExec}_2 : S_v
\end{array}
$$

The transition relation of the derived abstract machine is shown in Figures 4, 5, 6, and 7. Figure 4 defines the transitions interpreting expressions and tactics. Figure 5 defines the evaluation of function application to multiple arguments. We observe that the machine implements the eval/apply model of function application [26] that is inherited from the natural semantics. Figure 6 contains the transitions interpreting the stack controlling expression evaluation. Finally, Figure 7 displays the transitions interpreting the stack controlling tactic execution.

The initial configurations of the machine are of the form

$$\langle G, e, \mathsf{Nil}\rangle^{ee}_e$$

whereas the final configurations are of the form

$$\langle \mathsf{Nil}, r\rangle^x_a.$$

We state the correctness of the abstract machine with respect to the natural semantics:

THEOREM 1. *For any goal $G$ and closed expression $e$ we have:*

$$G \triangleright e \Downarrow r \quad \textit{iff} \quad \langle G, e, \mathsf{Nil}\rangle^{ee}_e \Rightarrow^* \langle \mathsf{Nil}, r\rangle^x_a,$$

*where $\Rightarrow^*$ denotes the reflexive-transitive closure of $\Rightarrow$.*

This theorem follows from the correctness of the functional correspondence, but it can also be established independently along the lines of the proof for the STG machine of Piróg and Biernacki [33].

In fact, through the functional correspondence, the abstract machine is not only *extensionally* but also *intensionally* equivalent with the natural semantics, i.e., the two *are different representations of the same evaluation model.* It follows that all design choices made at the level of the natural semantics are reflected in the abstract machine. Furthermore, any future change in the semantics of CoreLtac can be introduced at the level of the natural semantics and immediately accounted for in the abstract machine by the derivation method of the functional correspondence.

The abstract machine of this section directly corresponds to the natural semantics of CoreLtac and it has not been

$$
\begin{array}{lcl}
\langle \mathsf{Let}(G, x, e_2) : S_v, Gs \rangle_a^v & \Rightarrow & \langle S_v, \bot_0 \rangle_a^v \\
\langle \mathsf{Let}(G, x, e_2) : S_v, \bot_n \rangle_a^v & \Rightarrow & \langle S_v, \bot_n \rangle_a^v \\
\langle \mathsf{Let}(G, x, e_2) : S_v, v \rangle_a^v & \Rightarrow & \langle G, e_2[x := v], S_v \rangle_e^v
\end{array}
$$

$$
\begin{array}{lcl}
\langle \mathsf{Args}(G, es) : S_a, v \rangle_a^v & \Rightarrow & \langle G, es, \mathsf{Args}_2(v) : S_a \rangle_e^{as} \\
\langle \mathsf{Args}(G, es) : S_a, Gs \rangle_a^v & \Rightarrow & \langle S_a, \bot_0 \rangle_a^a \\
\langle \mathsf{Args}(G, es) : S_a, \bot_n \rangle_a^v & \Rightarrow & \langle S_a, \bot_n \rangle_a^a
\end{array}
$$

$$
\begin{array}{lcl}
\langle \mathsf{EExec}_1(G) : S_v, \bot_n \rangle_a^v & \Rightarrow & \langle S_v, \bot_n \rangle_a^v \\
\langle \mathsf{EExec}_1(G) : S_v, Gs \rangle_a^v & \Rightarrow & \langle S_v, Gs \rangle_a^v \\
\langle \mathsf{EExec}_1(G) : S_v, \lceil n \rceil \rangle_a^v & \Rightarrow & \langle S_v, \lceil n \rceil \rangle_a^v \\
\langle \mathsf{EExec}_1(G) : S_v, \lambda \vec{x}.e \rangle_a^v & \Rightarrow & \langle S_v, \lambda \vec{x}.e \rangle_a^v \\
\langle \mathsf{EExec}_1(G) : S_v, t \rangle_a^v & \Rightarrow & \langle G, t, \mathsf{EExec}_2 : S_v \rangle_e^x
\end{array}
$$

$$
\begin{array}{lcl}
\langle \mathsf{App}(G, es) : S_v, \bot_n \rangle_a^v & \Rightarrow & \langle S_v, \bot_n \rangle_a^v \\
\langle \mathsf{App}(G, es) : S_v, Gs \rangle_a^v & \Rightarrow & \langle S_v, \bot_0 \rangle_a^v \\
\langle \mathsf{App}(G, es) : S_v, \lceil n \rceil \rangle_a^v & \Rightarrow & \langle S_v, \bot_0 \rangle_a^v \\
\langle \mathsf{App}(G, es) : S_v, t \rangle_a^v & \Rightarrow & \langle S_v, \bot_0 \rangle_a^v \\
\langle \mathsf{App}(G, es) : S_v, \lambda \vec{x}.e \rangle_a^v & \Rightarrow & \langle G, es, \mathsf{Args}_1(G, \lambda \vec{x}.e) : S_v \rangle_e^{as}
\end{array}
$$

$$
\begin{array}{lcl}
\langle \mathsf{Pat}(G, m, e, \vec{cl}) : S_v, v \rangle_a^v & \Rightarrow & \langle S_v, v \rangle_a^v \\
\langle \mathsf{Pat}(G, m, e, \vec{cl}) : S_v, Gs \rangle_a^v & \Rightarrow & \langle S_v, Gs \rangle_a^v \\
\langle \mathsf{Pat}(G, m, e, \vec{cl}) : S_v, \bot_0 \rangle_a^v & \Rightarrow & \langle G, m, e, \vec{cl}, S_v \rangle_p \\
\langle \mathsf{Pat}(G, m, e, \vec{cl}) : S_v, \bot_{\mathsf{s}(n)} \rangle_a^v & \Rightarrow & \langle S_v, \bot_n \rangle_a^v
\end{array}
$$

$$
\begin{array}{lcl}
\langle \mathsf{EExpr}(G) : S_x, \bot_n \rangle_a^v & \Rightarrow & \langle S_x, \bot_n \rangle_a^x \\
\langle \mathsf{EExpr}(G) : S_x, Gs \rangle_a^v & \Rightarrow & \langle S_x, Gs \rangle_a^x \\
\langle \mathsf{EExpr}(G) : S_x, \lceil n \rceil \rangle_a^v & \Rightarrow & \langle S_x, \bot_0 \rangle_a^x \\
\langle \mathsf{EExpr}(G) : S_x, \lambda \vec{x}.e \rangle_a^v & \Rightarrow & \langle S_x, \bot_0 \rangle_a^x \\
\langle \mathsf{EExpr}(G) : S_x, t \rangle_a^v & \Rightarrow & \langle G, t, S_x \rangle_e^x
\end{array}
$$

**Figure 6:** Abstract machine – evaluation stack

$$
\begin{array}{lcll}
\langle \mathsf{Prog}(G) : S_x, \bot_n \rangle_a^x & \Rightarrow & \langle S_x, \bot_n \rangle_a^x & \\
\langle \mathsf{Prog}(G) : S_x, Gs \rangle_a^x & \Rightarrow & \langle S_x, \bot_0 \rangle_a^x & \text{if } Gs = [G] \\
\langle \mathsf{Prog}(G) : S_x, Gs \rangle_a^x & \Rightarrow & \langle S_x, Gs \rangle_a^x & \text{if } Gs \neq [G]
\end{array}
$$

$$
\begin{array}{lcl}
\langle \mathsf{Semi}(e) : S_x, \bot_n \rangle_a^x & \Rightarrow & \langle S_x, \bot_n \rangle_a^x \\
\langle \mathsf{Semi}(e) : S_x, Gs \rangle_a^x & \Rightarrow & \langle Gs, e, S_x \rangle_s
\end{array}
$$

$$
\begin{array}{lcl}
\langle \mathsf{Seq}_1(Gs, e) : S_x, \bot_n \rangle_a^x & \Rightarrow & \langle S_x, \bot_n \rangle_a^x \\
\langle \mathsf{Seq}_1(Gs, e) : S_x, Gs' \rangle_a^x & \Rightarrow & \langle Gs, e, \mathsf{Seq}_2(Gs') : S_x \rangle_s \\
\langle \mathsf{EExec}_2 : S_v, r \rangle_a^x & \Rightarrow & \langle S_v, r \rangle_a^v
\end{array}
$$

$$
\begin{array}{lcl}
\langle \mathsf{First}(G, es) : S_x, \bot_0 \rangle_a^x & \Rightarrow & \langle G, \mathsf{first}\ es, S_x \rangle_e^x \\
\langle \mathsf{First}(G, es) : S_x, \bot_{\mathsf{s}(n)} \rangle_a^x & \Rightarrow & \langle S_x, \bot_n \rangle_a^x \\
\langle \mathsf{First}(G, es) : S_x, Gs \rangle_a^x & \Rightarrow & \langle S_x, Gs \rangle_a^x
\end{array}
$$

$$
\begin{array}{lcl}
\langle \mathsf{Rep}(G, e) : S_x, \bot_0 \rangle_a^x & \Rightarrow & \langle S_x, [G] \rangle_a^x \\
\langle \mathsf{Rep}(G, e) : S_x, \bot_{\mathsf{s}(n)} \rangle_a^x & \Rightarrow & \langle S_x, \bot_n \rangle_a^x \\
\langle \mathsf{Rep}(G, e) : S_x, Gs \rangle_a^x & \Rightarrow & \langle Gs, \mathsf{repeat}\ e, S_x \rangle_s
\end{array}
$$

$$
\begin{array}{lcl}
\langle \mathsf{Seq}_2(Gs) : S_x, \bot_n \rangle_a^x & \Rightarrow & \langle S_x, \bot_n \rangle_a^x \\
\langle \mathsf{Seq}_2(Gs') : S_x, Gs'' \rangle_a^x & \Rightarrow & \langle S_x, Gs' \oplus Gs'' \rangle_a^x
\end{array}
$$

**Figure 7:** Abstract machine – execution stack

optimized in any way. However, at least two optimizations are possible. First of all, we could replace substitution with environments to make the process of function application more efficient, as is traditional in the design of abstract machines for functional languages [17]. Second of all, we could handle failures $\bot_n$ much more efficiently by re-designing the stacks of the abstract machine in a way resembling typical architecture of an abstract machine for exceptions or for delimited continuations [4], where the presence of a meta-stack (a stack of stacks) supports handling jumps.

# 6. REDUCTION SEMANTICS

Reduction semantics is a small-step operational semantics with explicit representation of contexts and a notion of reduction that characterizes basic steps of computation.

In this section we present a calculus of closures built on top of CoreLtac and we present its reduction semantics that faithfully accounts for CoreLtac.

The development is carried out along the lines of previous work of Biernacka and Danvy on the syntactic correspondence [5] and it consists in first defining a language with closures in which the intended reduction strategy can be represented, and then deriving an abstract machine using the refocusing procedure. In the present case, we observe that the machine for the language of closures can be transformed by short-circuiting redundant transitions and unfolding closures if we only want to operate on CoreLtac expressions and not on all closures. As a result we obtain a machine that coincides with the machine of Figures 4, 5, 6, and 7.

In CoreLtac, a computation is done in the context of a goal, therefore we introduce new syntactic categories of goal closures for each of the CoreLtac syntactic categories of Section 3. In order to be able to express single steps of computation of CoreLtac, the resulting calculus of closures introduces some auxiliary closures. The notion of reduction is defined by a separate relation for each mode.

The grammar of closures is as follows:

$$
\begin{array}{rrcl}
\text{(expression closure)} & c & ::= & r_v \\
 & & | & G \triangleright e \\
 & & | & c\ cs \\
 & & | & \mathsf{let}\ x := c_1\ \mathsf{in}\ c_2 \\
 & & | & \mathsf{mgoal}\ (m, c, \vec{cl}) \\
 & & | & \mathsf{meval}\ (m, \mathsf{pat}\ (G, e) \triangleright c, \vec{cl}) \\
 & & | & \mathsf{eval}\ ct \\
 & & | & \mathsf{exec}_1\ G \triangleright c \\
\text{(list closure)} & cs & ::= & \varepsilon \mid G \triangleright es \mid (c : cs) \mid \bot_n \\
\text{(tactic closure)} & ct & ::= & r_x \\
 & & | & \mathsf{exec}\ G \triangleright e \\
 & & | & G \triangleright t \\
 & & | & ct; e \\
 & & | & \mathsf{seq}\ s \\
 & & | & \mathsf{repeat}\ (G, e)\ ct \\
 & & | & \mathsf{first}\ (ct, \vec{c}) \\
 & & | & \mathsf{progress}\ G\ ct
\end{array}
$$

$$
\begin{array}{lrcl}
(\text{beta}_v) & (G \triangleright \lambda\vec{x}.e)\ \vec{v} & \to_v & \begin{cases} |\vec{v}| < |\vec{x}| \to \underline{\mathsf{let}}\ \ (\vec{x_1}, \vec{x_2}) = \mathsf{split}\ \vec{x}\ \mathsf{at}\ |\vec{v}|\ \ \underline{\mathsf{in}}\ \lambda\vec{x_2}.e[\overline{\vec{x_1} := \vec{v}}] \\ |\vec{v}| = |\vec{x}| \to G \triangleright e[\overline{\vec{x} := \vec{v}}] \\ |\vec{v}| > |\vec{x}| \to \underline{\mathsf{let}}\ \ (\vec{v_1}, \vec{v_2}) = \mathsf{split}\ \vec{v}\ \mathsf{at}\ |\vec{x}|\ \ \underline{\mathsf{in}}\ G \triangleright e[\overline{\vec{x} := \vec{v_1}}]\ \vec{v_2} \end{cases} \\
(\text{beta\_bot}) & (G \triangleright \lambda\vec{x}.e)\ \bot_n & \to_v & \bot_n \\
(\text{prop\_app}) & G \triangleright (e\ es) & \to_v & (G \triangleright e)\ (G \triangleright es) \\
(\text{app\_l\_bot}) & \bot_n\ cs & \to_v & \bot_n \\
(\text{app\_l\_nval}) & c\ cs & \to_v & \bot_0 \quad \text{if } c \neq \bot_n \text{ and } c \neq \lambda\vec{x}.e \\
(\text{prop\_let}) & G \triangleright \mathsf{let}\ x := e_1\ \mathsf{in}\ e_2 & \to_v & \mathsf{let}\ x := G \triangleright e_1\ \mathsf{in}\ G \triangleright e_2 \\
(\text{let\_gs}) & \mathsf{let}\ x := Gs\ \mathsf{in}\ c & \to_v & \bot_0 \\
(\text{let\_bot}) & \mathsf{let}\ x := \bot_n\ \mathsf{in}\ c & \to_v & \bot_n \\
(\text{let\_v}) & \mathsf{let}\ x := v\ \mathsf{in}\ c & \to_v & c[x := v] \\
(\text{mg\_nil}) & G \triangleright \mathsf{mgoal}\ \varepsilon & \to_v & \bot_0 \\
(\text{prop\_mg}) & G \triangleright \mathsf{mgoal}\ ((p, e) : \vec{cl}) & \to_v & \mathsf{mgoal}\ (\mathsf{start}\ G\ p, G \triangleright e, \vec{cl}) \\
(\text{mg\_none}) & \mathsf{mgoal}\ (m, G \triangleright e, \vec{cl}) & \to_v & G \triangleright \mathsf{mgoal}\ \vec{cl} \quad \text{if } \mathsf{next}\ m = \mathsf{Done} \\
(\text{mg\_match}) & \mathsf{mgoal}\ (m, G \triangleright e, \vec{cl}) & \to_v & \mathsf{meval}\ (m', \mathsf{pat}\ (G, e) \triangleright (\mathsf{exec}_1\ G \triangleright (G \triangleright \sigma(e))), \vec{cl}) \\
 & & & \quad \text{if } \mathsf{next}\ m = \mathsf{Match}(\sigma, m') \\
(\text{mg\_val}) & \mathsf{meval}\ (m', \mathsf{pat}\ (G, e)) \triangleright v, \vec{cl}) & \to_v & v \\
(\text{mg\_gs}) & \mathsf{meval}\ (m', \mathsf{pat}\ (G, e) \triangleright Gs, \vec{cl}) & \to_v & Gs \\
(\text{mg\_bot0}) & \mathsf{meval}\ (m', \mathsf{pat}\ (G, e) \triangleright \bot_0, \vec{cl}) & \to_v & \mathsf{mgoal}\ (m', G \triangleright e, \vec{cl}) \\
(\text{mg\_botS}) & \mathsf{meval}\ (m', \mathsf{pat}\ (G, e) \triangleright \bot_{S\,n}, \vec{cl}) & \to_v & \bot_n \\
(\text{exec1\_tac}) & \mathsf{exec}_1\ G \triangleright t & \to_v & \mathsf{eval}\ (G \triangleright t) \\
(\text{exec1\_res}) & \mathsf{exec}_1\ G \triangleright r_x & \to_v & r_x \\
(\text{exec1\_val}) & \mathsf{exec}_1\ G \triangleright v & \to_v & v \quad \text{if } v \neq t \\
(\text{eval\_res}) & \mathsf{eval}\ r_x & \to_v & r_x \\
(\text{goal\_val}) & G \triangleright v & \to_v & v
\end{array}
$$

**Figure 8:** Contraction rules for CoreLtac with closures – evaluation

$$
\begin{array}{rrcl}
\text{(sequence)} & s & ::= & \varepsilon \mid Gs \triangleright e \mid (ct : s) \\
\text{(evaluation result)} & r_v & ::= & r_x \mid v_{eval} \\
\text{(execution result)} & r_x & ::= & Gs \mid \bot_n
\end{array}
$$

The grammar of closures allows propagation of a goal inside a term in order to make it possible to compose intermediate results of computation. In addition, the results of execution (newly generated goals $Gs$ and the signal of error $\bot_n$) now become part of the syntax. As a consequence, execution errors can now also be propagated through single-step reductions. Moreover, we include "conversion" closures of the form $\mathsf{eval}\ ct$, $\mathsf{exec}\ G \triangleright e$ and $\mathsf{exec}_1\ G \triangleright c$ that serve to make transitions from one computation mode to the other. Specifically, $\mathsf{eval}\ ct$ denotes a closure that is first executed and then the result is used in the evaluation mode, in $\mathsf{exec}\ G \triangleright e$ the closure is first evaluated and if the result is a tactical, then it is executed. $\mathsf{exec}_1\ G \triangleright c$ is used only when evaluating clauses of the mgoal construct.

We have the following reduction contexts:

$$
\begin{array}{rrcl}
\text{(evaluation context)} & E_v & ::= & E_v[\mathsf{let}\ x := [\,]\ \mathsf{in}\ c] \\
 & & \mid & E_v[[\,]\ cs] \\
 & & \mid & E_a[[\,] : cs] \\
 & & \mid & E_v[\mathsf{mgoal}\ (m, [\,], \vec{cl})] \\
 & & \mid & E_v[\mathsf{exec}_1\ G \triangleright [\,]] \\
 & & \mid & E_x[\mathsf{exec}\ G \triangleright [\,]] \\
\text{(argument context)} & E_a & ::= & E_v[G \triangleright (v_{eval}\ [\,])] \\
 & & \mid & E_a[v_{eval} : [\,]]
\end{array}
$$

$$
\begin{array}{rrcl}
\text{(execution context)} & E_x & ::= & [\,] \\
 & & \mid & E_x[\mathsf{progress}\ G\ [\,]] \\
 & & \mid & E_x[\mathsf{first}\ ([\,] : cs)] \\
 & & \mid & E_x[[\,]; e] \\
 & & \mid & E_x[\mathsf{repeat}\ (G, e)\ [\,]] \\
 & & \mid & E_x[[\,]; s] \\
 & & \mid & E_x[Gs \triangleright [\,]] \\
 & & \mid & E_v[\mathsf{eval}\ [\,]]
\end{array}
$$

Echoing the previous semantics, we have three types of reduction contexts: evaluation-mode contexts $E_v$, execution-mode contexts $E_x$, and an auxiliary context $E_a$ for evaluation in an argument list (with appropriate markers indicating transitions between modes). Contexts correspond one-to-one to the stacks that appear in the abstract machine of the previous section, but here they are presented as "terms with a hole."

A single step of computation in a reduction semantics consists of the following 3 operations:

1. *decompose* the expression into a redex and a context
2. *contract* the redex
3. *plug* the contractum back into the context

This procedure is iterated until a result (here, a value or an error) or a stuck term is reached.

We omit the functions for decomposition and plugging from the presentation due to lack of space and present only the contraction rules in Figures 8 and 9.

$$
\begin{array}{lrcl}
\text{(idtac)} & G \triangleright \mathsf{idtac} & \to_x & [G] \\
\text{(fail)} & G \triangleright \mathsf{fail}\ n & \to_x & \bot_n \\
\text{(prop\_progress)} & G \triangleright \mathsf{progress}\ e & \to_x & \mathsf{progress}\ G\ (\mathsf{exec}\ G \triangleright e) \\
\text{(progress\_eq)} & \mathsf{progress}\ G\ Gs & \to_x & \bot_0 \quad \text{if } [G] = Gs \\
\text{(progress\_neq)} & \mathsf{progress}\ G\ Gs & \to_x & Gs \quad \text{if } [G] \neq Gs \\
\text{(progress\_bot)} & \mathsf{progress}\ G\ \bot_n & \to_x & \bot_n \\
\text{(prop\_repeat)} & G \triangleright \mathsf{repeat}\ e & \to_x & \mathsf{repeat}\ (G, e)\ (\mathsf{progress}\ G\ (\mathsf{exec}\ G \triangleright e)) \\
\text{(repeat\_bot0)} & \mathsf{repeat}\ (G, e)\ \bot_0 & \to_x & [G] \\
\text{(repeat\_botS)} & \mathsf{repeat}\ (G, e)\ \bot_{S\,n} & \to_x & \bot_n \\
\text{(repeat\_gs)} & \mathsf{repeat}\ (G, e)\ Gs & \to_x & Gs \triangleright \mathsf{repeat}\ e \\
\text{(seq\_bot\_l)} & \bot_n; e & \to_x & \bot_n \\
\text{(seq\_gs)} & Gs; e & \to_x & Gs \triangleright e \\
\text{(seq\_app)} & Gs; Gs' & \to_x & Gs \oplus Gs' \\
\text{(seq\_bot\_r)} & Gs; \bot_n & \to_x & \bot_n \\
\text{(prop\_semi)} & G \triangleright e_1; e_2 & \to_x & (\mathsf{exec}\ G \triangleright e_1); e_2 \\
\text{(prop\_first\_nil)} & G \triangleright \mathsf{first}\ \varepsilon & \to_x & \bot_0 \\
\text{(prop\_first\_cons)} & G \triangleright \mathsf{first}\ (e : es) & \to_x & \mathsf{first}\ \ (\mathsf{exec}\ G \triangleright e, G \triangleright es) \\
\text{(first\_bot0)} & \mathsf{first}\ \ (\bot_0, G \triangleright es) & \to_x & G \triangleright \mathsf{first}\ es \\
\text{(first\_botS)} & \mathsf{first}\ \ (\bot_{S\,n}, G \triangleright es) & \to_x & \bot_n \\
\text{(first\_gs)} & \mathsf{first}\ \ (Gs, G \triangleright es) & \to_x & Gs \\
\text{(execg\_res)} & \mathsf{exec}\ G \triangleright r_x & \to_x & r_x \\
\text{(execg\_val)} & \mathsf{exec}\ G \triangleright v & \to_x & \bot_0 \\
\text{(execg\_tac)} & \mathsf{exec}\ G \triangleright t & \to_x & G \triangleright t \\
\\
\text{(prop\_args\_nil)} & G \triangleright \varepsilon & \to_a & \varepsilon \\
\text{(prop\_args\_cons)} & G \triangleright (e : es) & \to_a & (G \triangleright e) : (G \triangleright es) \\
\text{(args\_goals)} & Gs : (G \triangleright es) & \to_a & \bot_0 \\
\text{(args\_bot)} & \bot_n : (G \triangleright es) & \to_a & \bot_n \\
\text{(varg\_bot)} & v : \bot_n & \to_a & \bot_n \\
\\
\text{(seq\_end)} & \varepsilon \triangleright e & \to_s & \varepsilon \\
\text{(seq\_cons)} & (G : Gs) \triangleright e & \to_s & (\mathsf{exec}\ G \triangleright e); (Gs \triangleright e)
\end{array}
$$

**Figure 9:** Contraction rules for CoreLtac with closures – execution and auxiliary contractions

We have already seen in the natural semantics that a dynamic semantics for CoreLtac interleaves computation in two modes. This was exemplified by indexing the judgment with a mode. We use the same approach here: we have *evaluations* of the form $E_v[c_1] \to E_v[c_2]$ if $c_1 \to_v c_2$ and *executions* of the form $E_x[ct_1] \to E_x[ct_2]$ if $ct_1 \to_x ct_2$. We also use auxiliary reductions $\to_a$ and $\to_s$ to process lists of arguments and sequences of goals and we have $E_a[cs] \to E_a[cs']$ if $cs \to_a cs'$ and $E_x[\mathsf{seq}\ s] \to E_x[\mathsf{seq}\ s']$ if $s \to_s s'$. The one-step reduction relation $\to$ is thus the compatible closure of all the types of contraction.

The reduction semantics is deterministic. The key lemma is the unique decomposition property:

LEMMA 1 (UNIQUE DECOMPOSITION). *Each expression closure is either a result or it can be uniquely decomposed into a potential redex (either a true redex as defined by contraction, or a stuck expression) and a reduction context (either an eval-context, an exec-context or an args-context).*

We state the correctness of the reduction semantics with respect to the abstract machine:

THEOREM 2. *For any goal $G$ and closed expression $e$ we have:*

$$\mathsf{exec}\ G \triangleright e \to^* r \quad \textit{iff} \quad \langle G, e, \mathsf{Nil} \rangle_e^{ee} \Rightarrow^* \langle \mathsf{Nil}, r \rangle_a^x,$$

*where $\to^*$ is the reflexive-transitive closure of $\to$.*

# 7. EXTENSIONS

## 7.1 Interaction with Gallina

An interesting feature of Ltac is the possibility of injecting Gallina terms ($tm$) in the tactic language. Ltac has a typeof $tm$ construction, which computes the type of term $tm$. Another useful feature is the ability to normalize a given (Gallina) term using eval $redexpr$ in $tm$, where $redexpr$ allows to choose, for instance, which evaluation strategy to use (cbv, lazy). We could add these two constructions to our language by the following rules:

$$\frac{G \vdash_{\mathsf{Gallina}} tm : tp}{G \triangleright \mathsf{constr}{:}\ tm \downarrow_v \mathsf{constr}{:}\ tm}$$

$$\frac{\neg(G \vdash_{\mathsf{Gallina}} tm : tp)}{G \triangleright \mathsf{constr}{:}\ tm \downarrow_v \bot_0}$$

$$\frac{G \triangleright e \downarrow_x \bot_n}{G \triangleright \mathsf{typeof}\ e \downarrow_x \bot_n}$$

$$\frac{G \triangleright e \downarrow_x \mathsf{constr}{:}\ tm \quad G \vdash_{\mathsf{Gallina}} tm : tp}{G \triangleright \mathsf{typeof}\ e \downarrow_x \mathsf{constr}{:}\ tp}$$

$$\frac{G \triangleright e \downarrow_x \bot_n}{G \triangleright \mathsf{eval}\ redexpr\ \mathsf{in}\ e \downarrow_x \bot_n}$$

$$\frac{G \triangleright e \downarrow_x \mathsf{constr}: tm \qquad \mathsf{Normalize}_{\mathsf{Gallina}}\ tm\ redexpr = tm_{nf}}{G \triangleright \mathsf{eval}\ redexpr\ \mathsf{in}\ e \downarrow_x \mathsf{constr}: tm_{nf}}$$

Terms of Gallina extend the category of values, but the evaluation rule has to invoke the internal typechecker to verify that the constructed term is well formed. This is because Ltac is dynamically typed.

## 7.2 More tactics

While we have kept CoreLtac as minimal as possible, it would not be a problem to extend it to support a larger set of Ltac's constructions. For instance, once we have introduced the ability to construct terms of Gallina, we can easily add the term matching construction (match $e$ with). We would need a different implementation of the matching abstraction, but apart from that, the rules would be the same as for match goal. Similarly, we could extend the pattern matching functionality with the context patterns, which allow one to match on all subterms on the given form – apart from syntactic matters, this also would require changes only in the matching abstraction. The same applies for match reverse goal.

A number of tacticals can be seen as syntactic sugar, e.g. try, solve, $e_1 \,||\, e_2$ can all be explained in terms of first and fail. Other tacticals can be seen as close variants of the presented ones: e.g. do $n$ $e$ is similar to repeat $e$ and the semantics for $e; [e_1|e_2|...|e_n]$ (the branching composition operator from LCF) can be easily obtained from the rules for $e_1; e_2$.

# 8. RELATED WORK

There exist a number of articles on the formal semantics of tactics, tacticals and tactic programming languages [27, 8, 2, 36, 13], but in this section we briefly review only the literature concerning Ltac.

In 1979, Gordon et al. [16] described the tactics and tacticals of Edinburgh LCF. The former are mostly inverted inference rules of the underlying logic (explained semi-formally as transformation of goals into subgoals) while the latter are defined by their implementation in ML.

20 years later, Delahaye designed Ltac, a Turing-complete domain specific language, to provide an intermediate ground between the limited capabilities of the tactic language available at the time in Coq (v.6.3, c. 2000) and the burden of programming tactics directly in the meta-language (which in case of Coq is OCaml [30]). He reported in [11, 12] that the rewrite of one of the decision procedures resulted in a significant gain in performance along with a dramatic decrease in the code size. This success was the consequence of the powerful backtracking behavior of the matching constructions that he introduced. He has only given an informal big-step semantics for Ltac.

Kirchner proposed a small-step operational semantics for Ltac in the form of a reduction semantics in [18]. The difference with our version is that he has assumed a rather complex interface of proof context objects and uses fairly complicated side conditions, while ours is rather abstract and thus perhaps more user-friendly. Moreover, he gives a simplified account for the exception mechanism.

Finally, it should be noted that the matching constructions as originally designed by Delahaye and described by Kirchner implemented backtracking only for the matchings of a single pattern. If the tactic from the right-hand side of the clause failed for all possible instantiations, then the whole expression failed. In the recent versions of Coq, other clauses are tried in such a case (unless the failure had a positive level).

# 9. CONCLUSION AND PERSPECTIVES

This article proposes an operational foundation for the tactic language of Coq. To the best of our knowledge, none of the existing work explains the failure raising and failure handling behavior inside Ltac in such detail. The abstract machine we presented appears to be the first abstract machine for tactic execution in the literature. Furthermore, all three semantic formats that we introduced are interconnected by known derivation methods and equivalent by construction. Consequently, any future extensions and modifications of the natural semantics described in this article will be mechanically accounted for in each of the remaining semantics.

We identify several directions for future work. First, we plan to define a formal semantics of a bigger subset of Ltac, and in particular to cover atomic tactics and more importantly, to treat some subtleties related to the dependent type theory of Coq, such as existential variables [3]. Second, we would like to consider applications of each of the presented semantic formats, including: a notion of equivalence of tactics and sound and complete reasoning techniques (natural or reduction semantics), tracing and debugging of proof scripts (reduction semantics), and semantic-based implementation (an optimized abstract machine).

Another line of work that we are going to pursue is a design, prescribed by a formal semantics, of a tactic language cleaner than Ltac, where evaluation of expressions and execution of tactics would be separated in a clear way. Such a separation could be supported by a lightweight type system with modalities, as is usual in the presence of staged computation [10]. As a matter of fact, designing a user-friendly type system for Ltac, statically eliminating certain common type errors, is a worthwhile task in its own right.

### *Acknowledgments*

We thank Paweł Wieczorek and Łukasz Dąbek for discussions and their heroic late-night proofreading. Thanks are also due to the anonymous referees for their invaluable comments. This work has been supported by the Polish National Science Center, grant number DEC-011/03/B/ST6/00348.

# 10. REFERENCES

[1] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *PPDP 2003*, pp. 8–19, Uppsala, Sweden, 2003.

[2] D. Aspinall, E. Denney, and C. Lüth. Tactics for hierarchical proof. *Mathematics in Computer Science*, 3(3):309–330, 2010.

[3] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[4] M. Biernacka, D. Biernacki, and O. Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1(2:5):1–39, 2005.

[5] M. Biernacka and O. Danvy. A concrete framework for environment machines. *ACM Transactions on*

*Computational Logic*, 9(1):1–30, 2007.

[6] T. Bourke, M. Daum, G. Klein, and R. Kolanski. Challenges and experiences in managing large-scale proofs. In *ICICM*, pp. 32–48, Bremen, Germany, 2012.

[7] A. Chlipala. Certified programming with dependent types. `http://adam.chlipala.net/cpdt/`.

[8] C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli. Tinycals: Step by step tacticals. *ENTCS*, 174(2):125–142, 2007.

[9] O. Danvy and L. R. Nielsen. Syntactic theories in practice. In *RULE 2001*, *ENTCS*, 59(4), Firenze, Italy, 2001.

[10] R. Davies and F. Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001.

[11] D. Delahaye. A tactic language for the system Coq. In *LPAR 2000*, *LNCS* 1955, pp. 85–95, Reunion Island, 2000.

[12] D. Delahaye. A Proof Dedicated Meta-Language. In *LFM 2002*, *ENTCS*, 70(2), pp. 96–109, Copenhagen, Denmark, 2002.

[13] E. Denney, J. Power, and K. Tourlas. Hiproofs: A hierarchical notion of proof tree. *ENTCS*, 155, pp. 341–359, 2006.

[14] G. Gonthier. The four colour theorem: Engineering of a formal proof. In *ASCM 2007*, LNCS 5081, pp. 333–333. Singapore, 2007.

[15] G. Gonthier. Engineering mathematics: the odd order theorem proof. In *POPL 2013*, pp. 1–2, Rome, Italy, 2013.

[16] M. J. C. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, *LNCS* 78, 1979.

[17] T. Hardin, L. Maranget, and B. Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–172, 1998.

[18] F. Kirchner. Coq tacticals and PVS strategies: A small-step semantics. In *Design and Application of Strategies/Tactics in Higher Order Logics*, pp. 69–83, 2003.

[19] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers. In *ICFP 2005*, SIGPLAN Notices, Vol. 40, No. 9, pp. 192–203, Tallinn, Estonia, 2005.

[20] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *SIGOPS 2009*, pp. 207–220. 2009.

[21] J.-L. Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007.

[22] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[23] D. K. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of Standard ML. In *POPL 2007*, pp. 173–184, Nice, France, 2007.

[24] X. Leroy. The Zinc experiment: an economical implementation of the ML language. Rapport Technique 117, INRIA Rocquencourt, Le Chesnay, France, 1990.

[25] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL 2006*, pp. 42–54, Charleston, South Carolina, 2006.

[26] S. Marlow and S. L. Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *Journal of Functional Programming*, 16(4-5):415–449, 2006.

[27] A. P. Martin, H. B. Gardiner, and J. C. P. Woodcock. A tactic calculus – abridged version. *Formal Aspects of Computing*, 8(4):479–489, 1996.

[28] A. P. Martin and J. Gibbons. A monadic interpretation of tactics. Unpublished note, 2002.

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

[30] The Caml Language. `http://caml.inria.fr/`.

[31] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[32] F. Pfenning and C. Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In *CADE 1999*, *LNAI 1632*, pp. 202–206, Trento, Italy, 1999.

[33] M. Piróg and D. Biernacki. A systematic derivation of the STG machine verified in Coq. In Jeremy Gibbons, editor, *Haskell'10*, pp. 25–36, Baltimore, MD, 2010.

[34] J. C. Reynolds. Definitional interpreters for higher-order programming languages. Higher-Order and Symbolic Computation 11(4):363–397, 1998.

[35] The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.4pl2*, 2013.

[36] I. Whiteside, D. Aspinall, L. Dixon, and G. Grov. Towards formal proof script refactoring. In *ICICM 2011*, pp. 260–275, Bertinoro, Italy, 2011.