Operational Semantics of Ltac

A Formal Study of the Tactic Language of the Coq Proof Assistant

Wojciech Jedynak

A Master’s Thesis written under the supervision of Dariusz Biernacki

WROCLAW 2013
Abstract

In this thesis we investigate interactive theorem proving in the framework of operational semantics. Our main subject of study is Ltac, the tactic language of the popular Coq proof assistant, for which we provide a rigorous semantics. To put Ltac in a broader context, we provide formal explanations of various aspects of the proof engine, including the proof shell and the atomic tactics. Given this set-up, we present three formats of operational semantics for Ltac, 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 identify some shortcomings of the language which our semantics help to clarify and we attempt to address them by proposing a type system for Ltac.

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.
Acknowledgements

I am deeply indebted to my supervisor, Dariusz Biernacki, for his guidance throughout the years, for his unlimited patience with me and for the great passion from which I took inspiration. His support has been crucial for the completion of this thesis and I am most grateful for the massive amount of time he has dedicated to me. I would also like to thank Małgorzata Biernacka for the numerous hours spent on discussions on the thesis’s subject and for the tremendous effort that she put, together with Dariusz, into the article that formed the backbone of this thesis.

Thanks are also due to my friend, Paweł Wieczorek, who not only did stimulate my interest in type theory, but also has been a great companion during many projects. The heated discussions we had in Girard’s tavern were an outstanding routine. I would also like to thank Piotr Polesiuk and Łukasz Dąbek for the extraordinary collaboration we had (together with Pawel) during the 2013 ICFP contest. It was a fantastic break and – hopefully! – only the first such event to come.

In the end, nothing would have been possible without the love, encouragement and never-ending support from my Family. To all the folks that have been showing keen interest in my progress, I can finally say: It’s done!
Contents

I Introduction 1

1 The what, why and how 3
   1.1 The subject of study 3
   1.2 Background 3
      1.2.1 Proof assistants: from LCF to Coq 3
      1.2.2 Tactic languages: from tacticals to Ltac 4
   1.3 Motivations 4
   1.4 Goals 5
   1.5 The base of this thesis 5
   1.6 Contributions 6
   1.7 RoCoQo – a prototype proof engine 6
   1.8 Comparison with Coq 6
   1.9 Overview 7

II Tactics and Tacticals 9

2 The proof engine 11
   2.1 Components of a proof engine 11
   2.2 The workflow 12
   2.3 Formalization of the top level structure of the proof engine 14

3 Atomic tactics 17
   3.1 Natural deduction – a fundamental proof system 17
   3.2 Atomic tactics for Natural Deduction 18
   3.3 Correctness of atomic tactics and proof scripts 23

4 Proof representation 27
   4.1 Proof assistants and belief 27
   4.2 Typed lambda calculi for proof representation 28
      4.2.1 The simply typed lambda calculus: syntax and semantics 29
      4.2.2 Connection with natural deduction 30
   4.3 Tactics as proof builders 31
      4.3.1 Tactics and evidence. Partial proof builders 31
4.3.2 Validations in RoCoQo ........................................ 32
4.3.3 Correctness of atomic tactics revisited .................... 35

5 LCF tacticals .................................................... 37
  5.1 Semantics of tacticals ...................................... 37
  5.2 Common idioms and derived tactics ......................... 40
    5.2.1 Try .................................................. 40
    5.2.2 Solve ............................................... 40
    5.2.3 Repeat .............................................. 40
    5.2.4 Branching composition ................................ 41
  5.3 The composition operator vs. the dot command ............ 42
  5.4 Summary .................................................. 42

III Operational Foundation for Ltac  ........................................ 43

6 Ltac – the tactics ................................................ 45
  6.1 The semantics of failure based backtracking .............. 45
    6.1.1 Semantics of match goal ................................ 46
  6.2 Derived tactics revisited .................................. 49
    6.2.1 Alternation and failure trapping ...................... 49
    6.2.2 Updated repeat .................................... 50
  6.3 Small examples ............................................ 51
    6.3.1 Reimplementation of assumption ....................... 51
    6.3.2 Goal context cleanup ................................ 52
  6.4 Extended example: propositional reasoning ............... 52
  6.5 Summary .................................................. 53

7 Ltac – the expressions ........................................... 55
  7.1 Syntax of CoreLtac ........................................ 55
  7.2 Natural Semantics ......................................... 60
    7.2.1 The notion of modes ................................... 61
    7.2.2 The main judgments ................................... 61
    7.2.3 Composition of evaluation and execution ........... 61
    7.2.4 Other judgments ...................................... 62
    7.2.5 The let and match goal pitfall ....................... 63
  7.3 Examples .................................................. 65
    7.3.1 Simple objective, complex tactic ..................... 65
  7.4 Summary .................................................. 66

8 An abstract machine for Ltac .................................... 67
  8.1 Derivation of the machine .................................. 67
  8.2 Optimized abstract machine ................................ 69

9 A reduction semantics for Ltac .................................... 73
IV Typed Tactical Languages 79

10 A type system for Ltac 81
   10.1 Failures that originate from type errors . . . . . . . . . . . . 81
   10.2 Type system for expressions . . . . . . . . . . . . . . . . . . . 82
   10.3 A type system for abstract machine configurations . . . . . 83
   10.4 Type safety . . . . . . . . . . . . . . . . . . . . . . . . . . 87
   10.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88

V Conclusion and references 89

11 Conclusion 91
   11.1 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
   11.2 Perspectives . . . . . . . . . . . . . . . . . . . . . . . . . . 91

A Streszczenie 101
List of Figures

2.1 Proof shell essentials – proof engine top level ........... 16
3.1 Natural deduction ...................................................... 18
3.2 Atomic tactic execution – user view of successes ........... 20
3.3 Atomic tactic execution – failures ............................... 21
3.4 Translation from derivations to scripts ....................... 25
4.1 STLC – typing rules ................................................... 29
4.2 Signature for proof builders ....................................... 33
5.1 Execution of LCF tacticals ........................................... 39
6.1 Tactic execution – tacticals ......................................... 47
6.2 Tactic execution – match goal ...................................... 50
6.3 Tactic execution – progress ......................................... 50
7.1 Natural semantics – expression execution ..................... 57
7.2 Natural semantics – extended expression evaluation ......... 57
7.3 Natural semantics – tactic execution ............................ 58
7.4 Natural semantics – expression evaluation ..................... 59
7.5 Natural semantics – lazy match goal ......................... 64
8.1 Abstract machine – expression evaluation and tactic execution 70
8.2 Abstract machine – function application ..................... 71
8.3 Abstract machine – execution stack ............................ 71
8.4 Abstract machine – evaluation stack ............................. 72
9.1 Reduction semantics – evaluation ................................. 77
9.2 Reduction semantics – execution and auxiliary contractions 78
10.1 Type system – expressions and tactics ....................... 84
10.2 Type system – machine stacks .................................. 85
10.3 Type system – results ............................................... 86
10.4 Type system – configurations .................................... 86
Part I

Introduction
Chapter 1

The \textit{what, why and how}

1.1 The subject of study

The subject of this thesis is the semantics of the tactic language of Coq. Coq is a modern proof assistant, that is a computer program for modeling domains of interest in a formal framework (of the underlying logic of Coq) and for stating and proving properties of the model. While the user could in principle encode the proofs by writing direct proof objects, Coq, as most other proof assistants, exposes a more user-friendly interface in the form of tactics. Coq’s proof engine is pretty sophisticated and the user has access to Ltac, a tactic programming language. The idea is that tactics correspond to very small reasoning steps and it would be impractical to build big proofs in such a manner. Ltac allows the user to not only create her own tactics, possibly on a higher level of abstraction, but also to create highly automated decision procedures, capable of discharging (conceptually simple, but tedious to formalize) proof obligations.

In the last decade there has been an increase of interest in proof assistants because the tools have become more mature and some impressive developments have been completed. In case of Coq, arguably the most interesting projects have been the proof of the Four Color Theorem by Gonthier \cite{Gon07} and the certification of a compiler for a subset of C by Leroy: \cite{Ler06}. Recently, the completion of project to formalize the proof of the Odd Order Theorem of Group Theory, done by a team led by Gonthier, has gained a lot of interest \cite{Gon13}.

1.2 Background

1.2.1 Proof assistants: from LCF to Coq

LCF \cite{GMW79} is one of the oldest proof assistants and not only did it influence modern systems such as Coq \cite{BC04}, HOL \cite{GM93} and Isabelle
[NWP02], but it also introduced the use of parametric polymorphism and relied heavily on type inference. LCF’s logic has been embedded in ML (Meta Language) – a functional programming language, which is the precursor of Standard ML [AM91] and OCaml [Oca].

De Bruijn’s AUTOMATH [NGdV94] has introduced the use of the Curry-Howard Isomorphism [SU06] for proof assistants. This approach is still widely used, with Coq and Matita [Mat] being the main examples.

Many other proof assistants are still being actively developed or have been in the past. Wiedijk’s book [Wie06] presents the proof of the irrationality of $\sqrt{2}$ in 17 proof assistants. Harrison discusses many aspects of formalized proofs (procedural vs. imperative, highly automated vs. requiring user intervention, etc.) in [Har98], while Geuvers presents a history of proof assistants [Geu09]. The authors of Matita present the structure of their project and propose a comparison of the architectures employed by other proof assistant implementers [ACTZ07].

1.2.2 Tactic languages: from tacticals to Ltac

In 1979, Gordon et al. [GMW79] 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 tactical 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 [Oca]). He reported in [Del00, Del02] 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.

1.3 Motivations

Our motivations are twofold. The first is a didactic one. By a rigorous study of the engine of an interactive proof assistant we hope to give the reader a somewhat deeper understanding of the tool. As we will demonstrate in course of the text, a precise analysis of a system may uncover many corner cases and surprising interactions. We will also attempt to identify the drawbacks, pitfalls, idiosyncrasies and deficiencies of the present tactic language of Coq.

The second motivation is to prepare ground for further studies (including tool development) in the form of a complete reference for some aspects of Ltac. While some warts of a language may never affect the user in practice,
it is important to care about them while developing any non-trivial script manipulating applications. For instance, a tactic refactoring tool should be semantics preserving. Strictly speaking, since the state-of-the-art proof assistants verify the whole development, it is possible to check if the modified script still works (and to undo the changes should the script be broken by the tool). However, as reported by Bourke et al. [BDKK12] in a large-scale proof development such checks are very time-consuming,\(^1\) so this approach quickly becomes impractical. It would be therefore very useful to build certified (or verified) tools that the users could trust and which could improve the efficiency of the proof engineering process by a great deal. The aforementioned paper gives some further ideas.

### 1.4 Goals

So far (to the best of our knowledge) the only attempt to formalize the semantics of Ltac has been Kirchner’s reduction semantics [Kir03] published in 2003. Since then the language has been changed and an update is called for. More importantly, we believe that Kirchner’s semantic did not uncover some subtleties of Ltac. Coq’s reference manual [Tea13] contains many details and it sufficient for the casual user, but many corner cases are left unspecified. Without a complete and up-to-date formal reference, it is impossible to even dream about the design of tools mentioned in the previous section.

Therefore the first goal of the thesis is to describe precisely and formally the proof engine of a proof assistant, with focus on the semantics, i.e., the meaning, of the tacticals, thus providing a sound base for further study and a solid documentation for the users.

The second goal is to improve the state-of-the-art tactic languages by designing a practical type system that prevents certain kinds of erratic behavior.

### 1.5 The base of this thesis

This thesis (especially the part from Chapter 7 to Chapter 9) is based on the article by Wojciech Jedynak, Małgorzata Biernacka and Dariusz Biernacki entitled *An Operational Foundation for the Tactic Language of Coq* accepted for presentation at the 15th International Symposium on Principles and Practice of Declarative Programming (PPDP 2013) and published in the conference proceedings [JBB13].

\(^1\)They report that the complete check for the l4 verified project takes 8 hours.
1.6 Contributions

In this thesis we give an operational account for various aspects of interactive theorem proving. By using the framework of operational semantics we make sure that all the pieces fit together, while we can reason about each and every part of the proof engine separately.

We give a formal semantics for the proof engine. We demonstrate how to design a set of atomic tactics given a logical inference system and how to prove the correctness of the design. We also discuss the design of proof representations based on typed lambda calculi.

As our main contribution, we introduce CoreLtac, a subset of Ltac, capable of illustrating many interesting characteristics of the whole language. We give a natural semantics for CoreLtac and we uncover many subtleties of the language. We then study Ltac as a programming language and thus apply off-the-shelf derivational techniques to obtain an operational foundation for CoreLtac:

1. the functional correspondence [ABDM03] yields an abstract machine,
2. the syntactic correspondence [DN01] yields a reduction semantics.

Finally, we design a type system for CoreLtac and prove its correctness with respect to the operational semantics.

1.7 RoCoQo – a prototype proof engine

This thesis is accompanied by an implementation of a prototype proof engine. The software is intended to replicate many features present in Coq, is thus named RoCoQo and also implemented in OCaml. More concretely, RoCoQo's atomic tactics are as presented in Chapter 3. The tactic language implemented in RoCoQo is CoreLtac from Chapter 7 and the interpreter for CoreLtac is based on the natural semantics from the same chapter.

Why do we provide RoCoQo as a supplementary material for the thesis? A theorem prover deals with many details (such as proof term construction, proof state history management for backtracking purposes, etc.) and in our semantics we focus on certain aspects while abstracting other detail. Should the reader want to see the whole picture, verify the gory details, run and test the semantics, etc., she is welcome to inspect and compile the source code.

1.8 Comparison with Coq

We would like to warn the reader that this thesis is not intended as a full Coq tutorial and our main subject of study is Ltac, the tactical language
of Coq. We aimed to include the whole picture in thesis, so we describe the top-level proof engine and an example set of atomic tactics, however the latter is much simplified as compared to the large set of atomic tactics (more that 100 commands) present in Coq. Moreover, we use the simply typed lambda calculus as the internal logic, while Coq uses a much more sophisticated system based on the Calculus of Constructions [PPM90]. Our approach should illustrate that the ideas behind Ltac are universal and it would not be very difficult to implement a similar system in other proof assistants. As a consequence, the reader should treat the example tactics as illustrative examples rather than as snippets for direct copy-paste. For convenience we list the differences in a compact table below:

<table>
<thead>
<tr>
<th>Thesis</th>
<th>Coq</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit, ⊤</td>
<td>True</td>
</tr>
<tr>
<td>empty, ⊥ (as a formula)</td>
<td>False</td>
</tr>
<tr>
<td>$A \ast B$</td>
<td>$A \land B$</td>
</tr>
<tr>
<td>$A + B$</td>
<td>$A \lor B$</td>
</tr>
<tr>
<td>solve $t$</td>
<td>solve $t$</td>
</tr>
<tr>
<td>truth</td>
<td>constructor</td>
</tr>
</tbody>
</table>

### 1.9 Overview

The thesis is structured as follows. In Chapter 2 we describe the proof shell in a formal manner. In Chapter 3 we give a semantics for an example set of atomic tactics and in Chapter 4 we study proof representations. After that we begin to formalize various flavors of tactics and tacticals starting with the classic LCF in Chapter 5. We then extend the tactics with the match goal construction in Chapter 6 and finally in Chapter 7 we introduce CoreLtac, an interesting subset of Ltac. Starting with the natural semantics for CoreLtac, we derive an abstract machine (in Chapter 8) and a reduction semantics (in Chapter 9). In Chapter 10 we present a typing system for Ltac. In Chapter 11 we conclude.
Part II

Tactics and Tacticals
Chapter 2
The proof engine

This chapter gives the perspective for the whole thesis. It demonstrates the notions of a goal, tactic, and failure.

We begin with an introduction of the notions used throughout the thesis followed by a short example demonstrating the workflow. We present our prototype RoCoQo and formalize the proof engine’s main loop.

2.1 Components of a proof engine

A Proof context goal (or simply goal) is a theorem the user has to prove along with the context – a set of named hypotheses (introduced assumptions of the theorem). The proof engine’s state is a list (an ordered sequence) of goals. At first the list contains only the initial theorem that the user has provided. When this list becomes empty then the goal is solved and the theorem is proved. Otherwise, the first goal in the list (the current goal) is presented to the user and she is supposed to provide tactics, i.e., commands that modify the goal list. In almost all cases tactics break down the current goal into further goals (often called subgoals). The whole idea of interactive theorem proving relies on the fact that it is easier to prove theorems when they are divided into small, manageable parts. It should be noted that it is often convenient to identify in explanations the goal formula with the goal itself. We also often identify the type of the assumption with the assumption itself (so we forget about the name or identifier).

Not all tactic invocations succeed, they might fail in some cases. For example:

- **assumption** fails when none of the assumptions is equal to the goal theorem,

- **intro** fails when the goal theorem is not the arrow type (or, in a dependent type theory, a dependent product type).
For the sake of the discussion in the present chapter we assume that when a tactic fails, this fact is immediately reported to the user (possibly along with a descriptive message). In the sequel we will describe and study more sophisticated features such as failure (or error) trapping and failures with level tags.

2.2 The workflow

We now demonstrate interactive theorem proving by proving an example theorem in RoCoQo. During this example session we hope to make the reader accustomed to our terminology. A proper tutorial on interactive theorem proving is out of scope of the thesis, the interested reader should consult sources such as [Chl] or [BC04].

The interaction is marked by the use of the typewriter font. The text after the > character is the user’s input, everything else is the output of the tool.

We initialize the system and state the theorem we want to prove – the commutativity of conjunction (the asterisk denotes conjunction (or pairing, depending on your point of view)):

Rocoqo 0.19
> Goal (a * b) -> (b * a).

The system accepts the proposition and initializes the proof engine. We are told the number of goals and we see the current goal:

1 goal(s) remaining.
-----------------------------
Goal: ((a * b) -> (b * a))

The goal theorem is an implication so we call the introduction tactic:

> intro H.

1 goal(s) remaining.
H : (a * b)
-----------------------------
Goal: (b * a)

We have introduced an assumption named H. The horizontal line divides the context hypotheses (here only H) from the goal theorem.

The goal is a conjunction. The natural way to break it down is to show each conjunct separately. The split tactic does the breaking.
> split.

2 goal(s) remaining.
  H : (a * b)
  ------------------------
  Goal: b

We see that the split tactic has replaced our goal with 2 subgoals and only the first is shown. In a more complete system (such as Coq) we could check that in the other goal we have to show $a$.

The only way to proceed now is to extract the components of $H$:

> destruct H.

2 goal(s) remaining.
  q : b
  p : a
  H : (a * b)
  ------------------------
  Goal: b

The proof context has been extended with the components of the chosen pair. Since we have among our assumptions the goal theorem the current goal could now be solved by invoking trivial.

Wait, wouldn’t it have been better to destruct the pair when we still had a single goal? Now we will have to perform the extraction twice. Of course, all subgoals are still solvable, but the proof we will construct will be repetitive. Fortunately, most interactive theorem provers allow the user to backtrack:

> Undo.

2 goal(s) remaining.
  H : (a * b)
  ------------------------
  Goal: b

> Undo.

1 goal(s) remaining.
  H : (a * b)
  ------------------------
  Goal: (b * a)

So this time we first destruct and then split:
> destruct H.

1 goal(s) remaining.
q : b
p : a
H : (a * b)
-------------------------------
Goal: (b * a)

> split.

2 goal(s) remaining.
q : b
p : a
H : (a * b)
-------------------------------
Goal: b

We are almost done as both subgoals are now easy to finish:

> trivial.

Subgoal solved.
1 goal(s) remaining.
q : b
p : a
H : (a * b)
-------------------------------
Goal: a

> trivial.

Subgoal solved.
Proof completed.

The proof is finished. In principle, RoCoQo is now able to print the proof that we have just interactively constructed, but we omit this here as proof representations are discussed in Chapter 4.

2.3 Formalization of the top level structure of the proof engine

In this section we want to describe formally how does the goal list component of the proof engine’s state change while interacting with the user.
To model the interaction with user, we assume that we are given a proof script (i.e. a list of tactics (more generally: commands)) so we work in the batch mode. An example proof script in Coq’s notation:

\begin{verbatim}
intros. split. assumption. constructor. Qed.
\end{verbatim}

in our notation becomes \texttt{intros : split : assumption : constructor : ε} with ε denoting the empty list and \texttt{x : xs} denoting the consing of \texttt{x} to \texttt{xs}. We will also use \texttt{xs ⊕ ys} to denote list concatenation.

To describe the behavior of the top level of proof engine (sometimes called the proof shell) we use a small-step operational semantics in which a single reduction step consists of either processing of a single command or reaching a terminal state. The execution of a script \texttt{es} with the current goal stack \texttt{Gs} is described by the judgment

\[ Gs \triangleright es \rightarrow r \]

The input of the proof shell judgment is goal list and a tactic list. Both are destructed at the same time, which is highlighted visually in the semantics below by the use the same list notation for both types of lists. The output \texttt{r} is either an intermediate state \texttt{Gs' \triangleright es} with updated state and a shorter list of commands, or a value. Values are \texttt{✓} – a finished proof, \texttt{incomplete} – an incomplete proof, \texttt{err} – an indicator of a failing proof script.

Since we want to only describe the main loop of the proof shell, in this section we abstract over the execution of a single command. In fact, we will spend the remaining part of the thesis on the description of this very notion. Here we only want to put that work into perspective. We assume a judgment describing tactic execution of the form

\[ G \triangleright e \Downarrow r_{tac} \]

The result \texttt{r}_{tac} is either \texttt{Gs} – a list of subgoals generated by the tactic \texttt{e} or the error marker \texttt{⊥}.

We now discuss the semantic rules presented in Figure 2.1. When the whole script has been processed then either the goal stack is empty, denoting a successful proof (the \texttt{QED} rule) or some goals remain, so the proof is incomplete and the user should provide more tactics (the \texttt{STOP} rule). When the goal stack becomes empty, the proof shell terminates, so it is an error to provide a script that is too long (the \texttt{TOOMANY} rule). When both the goal and tactic stacks (lists) are non-empty, the top (first) elements are taken out and run using the tactic execution judgment. When the tactic fails the error is propagated\(^1\) (the \texttt{TAC-ERR} rule). Otherwise new subgoals are generated and

\(^1\)and reported to the user in an implementation.
they are put on the top of the stack (the \texttt{TAC-OK} rule).

With the transitive closure of the introduced notation (\(\rightarrow^+\)) we can formalize the notion we previously used informally:

\textbf{Definition 2.1.} A proof script \(es\) solves the goal \(G\) iff \((G: \varepsilon) \triangleright es\) \(\rightarrow^+\) \(\checkmark\)

We also prove a lemma that will prove useful later:

\textbf{Lemma 2.2} (concatenation lemma). The following rule is admissible:

\[
\text{\textbf{CONCAT} } \quad (Gs_1 \triangleright es_1) \rightarrow^+ \checkmark \\
(Gs_2 \triangleright es_2) \rightarrow^+ \checkmark
\]

\[
(Gs_1 \oplusGs_2) \triangleright (es_1 \oplus es_2) \rightarrow^+ \checkmark
\]

\textit{Proof.} Routine rule induction on the derivation of \((Gs_1 \triangleright es_1) \rightarrow^+ \checkmark\), using the following properties of append:

\[
\varepsilon \oplus l = l
\]

\[
l_1 \oplus (l_2 \oplus l_3) = (l_1 \oplus l_2) \oplus l_3
\]

Finally, we should state that if the tactic execution relation is functional and decidable then so is the proof shell relation.
Chapter 3

Atomic tactics

In this chapter we want to introduce the notion of atomic tactics. However, to do so we need to decide on a logic, so that we can talk about proofs and proof search.

3.1 Natural deduction – a fundamental proof system

We recall the natural deduction (ND), originally introduced by Gentzen [Pra65], for the intuitionistic propositional logic, using the sequent notation. The only twist is that in our case, the context (denoted Γ) is not a set of formulas (called assumptions), but a list of (name, formula) pairs, with each name distinct. Compared to the usual presentation this change adds a bit of technical complexity, but the benefits will come forth later. Above all, the user of a proof assistant has to reference assumptions and the natural way is to use names.

The syntax of formulas is

\[ \phi, \psi ::= p \mid \bot \mid \top \mid \phi \lor \phi \mid \phi \land \phi \mid \phi \Rightarrow \phi \]

where \( p \) represents an element of the countably infinite set of propositional variables.

The main judgment is

\[ \Gamma \vdash \phi \]

which is read as “the formula \( \phi \) is provable from \( \Gamma \).”

The inference rules are presented in Figure 3.1. The distinguishing feature of natural deduction is that all logical connectives are presented using introduction and elimination rules. The introduction rules explain what needs to be asserted to obtain a proof of the given connective, while elimination rules provide a mean to use (or decompose) each connective. However, the formula \( \top \) is always deducible, so no useful truth can be deduced from
it. Hence it has no elimination rule. Dually, ⊥ should never be deducible in a consistent context, so it has no introduction rule. This intuition justifies the elimination rule for ⊥, which has been traditionally called *ex falso quodlibet* – “anything follows from a contradiction.” Finally, we have the **HYP** rule, which is a structural rule, so it is required by the choice of the formalism. To ensure the invariant that all names in the context are distinct, rules that extend the context (namely ⇒I and ∨E) have implicit side conditions: the names of new hypotheses are assumed to be *fresh*.

We also mention two structural properties of the system (both proven by rule induction on the derivation):

**Lemma 3.1** (Weakening). \( \Gamma \vdash \phi \) implies \( \Gamma, x : \psi \vdash \phi \).

**Lemma 3.2** (Substitution principle). \( \Gamma, x : \psi \vdash \phi \) and \( \Gamma \vdash \psi \) imply \( \Gamma \vdash \phi \).
The resulting derivation is formed from the first assumption with the second assumption being substituted for any appeal to the **HYP** rule for \( x \).

For a more complete presentation of intuitionistic logic and natural deduction we refer the reader to Urzyczyn and Sørensen’s monograph [SU06].

### 3.2 Atomic tactics for Natural Deduction

An **atomic tactic** is a command for the proof engine that performs a step of goal directed (or backward) reasoning. The most basic way to obtain a set of useful tactics is to read the inference rules of the underlying logical
framework in the bottom-up direction. For instance, the split tactic is based on the $\land$ rule and it reduces a goal of the form $\phi_1 \land \phi_2$ into two (simpler) subgoals: $\phi_1$ and $\phi_2$, whereas the intro tactic (based on $\Rightarrow$) replaces $\phi_1 \Rightarrow \phi_2$ with assuming $\phi_1$. In some cases, it is better for usability to design tactics that correspond to derived or admissible inference rules. Many equivalent possibilities exist, for the sake of this section we have chosen a set of tactics inspired by the ones used in Coq.

The syntax of atomic tactics is

$$atac ::= \text{intro } x | \text{assumption } | \text{apply } H$$
$$\ | \text{left } | \text{right } | \text{split } | \text{truth}$$
$$\ | \text{destruct } H x y | \text{exfalso}$$
$$\ | \text{assert } x \phi$$

$H, x, y$ are concrete names of hypotheses. To ease memory, $H$ is used for hypotheses already present in the context, while $x, y$ denote fresh names to be introduced.

Recalling the example tactics from above, we notice that the (main) effect of a tactic is to change a goal into a list of subgoals. Two things can happen in this process:

- the goal theorem is changed (e.g. in split it is broken down into sub-components),
- new assumptions are introduced (e.g. intro).

Therefore, unlike the proof shell semantics, in the formal semantics of atomic tactics we must expose the internal structure of a goal, so instead of $G$ we write $\Gamma \div \phi$.

We begin the semantic presentation with a version suitable for the user, i.e., we postpone (until Section 4.3) the discussion of the details concerning the encoding of inferences made by the tactics.

The atomic tactic execution judgment is

$$(\Gamma \div \phi) \triangleright atac \downarrow r$$

The result $r$ is either a list of subgoals or the error marker $\bot$. We use the $[\Gamma \div \phi]$ notation to denote a singleton list.

Every atomic tactic (besides exfalso) has a non-trivial precondition. If the precondition is satisfied then the execution of the tactic is successful and results in a (possibly empty) list of subgoals. On the other hand, the execution fails when the precondition is not satisfied. Because the situation is rather black and white, we have divided the semantic rules into two figures: Figure 3.2 shows the successful cases, while Figure 3.3 presents the failures. The reader is encouraged to study the former while treating the latter figure as reference – the rules for failures are included to make the tactic execution relation total.
\[ \exists H. H : \phi \in \Gamma \]

\[ (\Gamma \div \phi) \triangleright \text{assumption} \downarrow \varepsilon \]

\[ \text{INTRO-OK} \quad \text{fresh} \ x \ \Gamma \]

\[ (\Gamma \div \psi \Rightarrow \phi) \triangleright \text{intro} \ x \downarrow \ [\Gamma, x : \psi \div \phi] \]

\[ \text{APPLY-OK} \quad H : \psi \in \Gamma \quad \psi = \phi_1 \Rightarrow \ldots \Rightarrow \phi_n \Rightarrow \phi \quad (n \geq 0) \]

\[ (\Gamma \div \phi) \triangleright \text{apply} \downarrow \ (\Gamma \div \phi) \downarrow \ldots \downarrow (\Gamma \div \phi) : \varepsilon \]

\[ \text{LEFT-OK} \quad (\Gamma \div \phi_1 \lor \phi_2) \triangleright \text{left} \downarrow [\Gamma \div \phi_1] \]

\[ \text{RIGHT-OK} \quad (\Gamma \div \phi_1 \lor \phi_2) \triangleright \text{right} \downarrow [\Gamma \div \phi_2] \]

\[ \text{SPLIT-OK} \quad (\Gamma \div \phi_1 \land \phi_2) \triangleright \text{split} \downarrow (\Gamma \div \phi_1) : (\Gamma \div \phi_2) : \varepsilon \]

\[ \text{TRUTH-OK} \quad (\Gamma \div \top) \triangleright \text{truth} \downarrow \varepsilon \]

\[ \text{DESTRUCT-OK}_1 \quad H : \psi_1 \land \psi_2 \in \Gamma \quad x \neq y \quad \text{fresh} \ x \ \Gamma \quad \text{fresh} \ y \ \Gamma \]

\[ (\Gamma \div \phi) \triangleright \text{destruct} \ H \ x \ y \downarrow [\Gamma, x : \psi_1, y : \psi_2 \div \phi] \]

\[ \text{DESTRUCT-OK}_2 \quad H : \psi_1 \lor \psi_2 \in \Gamma \quad x \neq y \quad \text{fresh} \ x \ \Gamma \quad \text{fresh} \ y \ \Gamma \]

\[ (\Gamma \div \phi) \triangleright \text{destruct} \ H \ x \ y \downarrow (\Gamma, x : \psi_1 \div \phi) : (\Gamma, y : \psi_2 \div \phi) : \varepsilon \]

\[ \text{EXFALSO-OK} \quad (\Gamma \div \phi) \triangleright \text{exfalso} \downarrow [\Gamma \div \bot] \]

\[ \text{ASSERT-OK} \quad \text{fresh} \ x \ \Gamma \]

\[ (\Gamma \div \phi) \triangleright \text{assert} \ x \ \psi \downarrow (\Gamma \div \psi) : (\Gamma, x : \psi \div \phi) : \varepsilon \]

**Figure 3.2:** Atomic tactic execution – user view of successes
### CHAPTER 3. ATOMIC TACTICS

#### Figure 3.3: Atomic tactic execution – failures

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premise</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSUMPTION-FAIL</td>
<td>¬(∃H. H : φ ∈ Γ)</td>
<td>(Γ ⊢ φ) ∴ assumption ↓ ⊥</td>
</tr>
<tr>
<td>INTRO-FAIL</td>
<td>¬(fresh x Γ) ∨ φ ≠ φ₁ ⇒ φ₂</td>
<td>(Γ ⊢ φ) ∴ intro x ↓ ⊥</td>
</tr>
<tr>
<td>APPLY-FAIL₁</td>
<td>fresh H Γ</td>
<td>(Γ ⊢ φ) ∴ apply H ↓ ⊥</td>
</tr>
<tr>
<td>APPLY-FAIL₂</td>
<td>H : ψ ∈ Γ ψ not compatible with φ</td>
<td>(Γ ⊢ φ) ∴ apply H ↓ ⊥</td>
</tr>
<tr>
<td>LEFT-FAIL</td>
<td>φ ≠ φ₁ ∨ φ₂</td>
<td>(Γ ⊢ φ) ∴ left ↓ ⊥</td>
</tr>
<tr>
<td>RIGHT-FAIL</td>
<td>φ ≠ φ₁ ∨ φ₂</td>
<td>(Γ ⊢ φ) ∴ right ↓ ⊥</td>
</tr>
<tr>
<td>SPLIT-FAIL</td>
<td>φ ≠ φ₁ ∧ φ₂</td>
<td>(Γ ⊢ φ) ∴ split ↓ ⊥</td>
</tr>
<tr>
<td>TRUTH-FAIL</td>
<td>φ ≠ ⊤</td>
<td>(Γ ⊢ φ) ∴ truth ↓ ⊥</td>
</tr>
<tr>
<td>DESTRUCT-FAIL₁</td>
<td>fresh H Γ ∨ x = y ∨ ¬(fresh x Γ) ∨ ¬(fresh y Γ)</td>
<td>(Γ ⊢ φ) ∴ destruct H x y ↓ ⊥</td>
</tr>
<tr>
<td>DESTRUCT-FAIL₂</td>
<td>H : ψ ∈ Γ ψ ≠ φ₁ ∧ φ₂ ψ ≠ φ₁ ∨ φ₂</td>
<td>(Γ ⊢ φ) ∴ destruct H x y ↓ ⊥</td>
</tr>
<tr>
<td>ASSERT-FAIL</td>
<td>¬(fresh x Γ)</td>
<td>(Γ ⊢ φ) ∴ assert x ψ ↓ ⊥</td>
</tr>
</tbody>
</table>
We now proceed to briefly comment on the tactics and their preconditions. Whenever a new name is introduced ($x$, $y$ in `intro`, `destruct` and `assert`) then its freshness must be checked, because the proof context must remain unambiguous. On the other hand, if the tactic’s argument is supposed to name a hypothesis ($H$ in `apply` and `destruct`), we must look up the type of the hypothesis. Therefore it is an error to provide a name unbound in the context. The `intro`, `split`, `left`, `right` and `truth` tactics correspond to introduction rules so they succeed only when the goal formula has the appropriate shape. The `destruct` tactic can be used to eliminate both conjunctions and disjunctions, however they have to be already present in the context as assumptions. When it is not the case, the `assert` tactic can be used to introduce a local lemma. In contrast, the `exfalso` tactic changes the goal into $\bot$.

It would be possible to unify those approaches: in Coq, the `destruct` tactic can handle any inductive type, but this requires the use of special syntax called `intro patterns`. We decided to rather use two tactics for destruction to shorten the presentation.

The tactic with arguably the most complicated precondition is `apply`, which corresponds to the modus ponens logical inference (the $\Rightarrow_E$ rule from Figure 3.1). The nuance is that a single invocation of `apply` can perform many applications of modus ponens and we must check if the assumption $\psi$ can be decomposed into a compound implication with final formula $\phi$, that is, whether

$$\psi = \phi_1 \Rightarrow (\phi_2 \Rightarrow (\ldots \Rightarrow (\phi_n \Rightarrow \phi)))$$

for some $\phi_1, \phi_2, \ldots, \phi_n$. If this is the case, then we say that $\psi$ and $\phi$ are compatible and `apply` generates $n$ subgoals — $\phi_1, \phi_2, \ldots, \phi_n$ (so each assumption of $\psi$ becomes a subgoal). Otherwise, the tactic fails. Finally, we should mention that in our presentation the names introduced by the tactics are expected to be provided by the user. The reason is technical: we can formalize and prove the correctness of such a tactic design – see next section. However, in practice this limitation may be a burden. Coq allows one to omit those names and will automatically generate a fresh identifier if needed. This approach has many advantages, many of which will become apparent in subsequent chapters, when we introduce tacticals and tactical languages. For now it suffices to realize that it is safe to call the bare `intro` twice in a row, but in case of `intro x`, the second invocation would fail because of the name clash. A dual aspect is automatic inference of hypotheses’ names. The `assumption` tactic corresponds to the `HYP` rule, but the user need

---

1. In the programming language interpretation (explained later), $H$ is a function available in the local scope. Suppose that $H$ has type $\text{int} \rightarrow \text{double} \rightarrow \text{char}$. If we are trying to construct a value of type `char` then $H$ can be used to this end, provided we can construct values of type `int` and `double`. Similarly, we could construct a value of type `double` $\rightarrow$ `char` (by providing a single argument to $H$ of type `int`). On the other hand, $H$ can not be used to (directly) construct values of types such as `float` or `int $\rightarrow$ double`. The `apply` tactic is able to perform this check automatically and it sets the required arguments as subgoals.
not provide the variable name. Perhaps surprisingly, \texttt{apply} can be used as an explicit variant of \texttt{assumption}, as the side condition is \(n \geq 0\), not \(n \geq 1\).

### 3.3 Correctness of atomic tactics and proof scripts

It is pretty clear that tactics that are just inference rules read bottom-up should be correct with respect to the underlying logic. What about more complicated tactics such as \texttt{apply} or \texttt{destruct}? The correctness can be stated formally as a pair of theorems:

**Theorem 3.3** (Soundness of scripts of atomic tactics).

\[
\Gamma \vdash \phi \Rightarrow \exists es. [\Gamma \vdash \phi] \Rightarrow \vdash^+ \checkmark
\]

**Theorem 3.4** (Completeness of scripts of atomic tactics).

\[
\Gamma \vdash \phi \Rightarrow \exists es. [\Gamma \vdash \phi] \Rightarrow es \Rightarrow^+ \checkmark
\]

Soundness states that scripts can only solve goals that correspond to provable sequents, while completeness states that all provable sequents can be proved interactively.

**Notations.** \([1,n]\) is the set of integers \(\{1, \ldots, n\}\). \([\Gamma_i \vdash \phi_i]_{i=1}^n\) denotes the list \([\Gamma_1 \vdash \phi_1, \ldots, \Gamma_n \vdash \phi_n]\).

To prove soundness we can split the work into two lemmas:

**Lemma 3.5** (Soundness of atomic tactics). If

\[
(\Gamma \vdash \phi) \Rightarrow atac \Downarrow [\Gamma_i \vdash \phi_i]_{i=1}^n
\]

and

\[
\forall i \in [1, n]. \Gamma_i \vdash \phi_i
\]

then

\[
\Gamma \vdash \phi
\]

**Proof.** By case analysis on \texttt{atac} and inversion. Most cases are immediate. In \texttt{apply} we need to use the \(\Rightarrow \text{E}\) rule \(n\) times, so a helper induction is required. For \texttt{assert} and \texttt{destruct} (in the \texttt{DESTRUCT-OK} case) we can create a detour using implication introduction followed by an immediate elimination or we can appeal to the substitution principle.

**Lemma 3.6** (Generalized soundness of scripts). If

\[
[\Gamma_i \vdash \phi_i]_{i=1}^n \Rightarrow es \Rightarrow^+ \checkmark
\]

then

\[
\forall i \in [1, n]. \Gamma_i \vdash \phi_i
\]
Proof. By rule induction. The \texttt{QED} case is immediate. In the \texttt{TAC-OK} case the conclusion \((G : Gs) \vdash (atac : es) \rightarrow^+ \checkmark\) is decomposed into

\[
\begin{array}{c}
G \vdash atac \downarrow Gs' \\
\hline
(G : Gs) \vdash (atac : es) \rightarrow (Gs' \oplus Gs) \vdash es
\end{array}
\]

and \((Gs' \oplus Gs) \vdash es \rightarrow^+ \checkmark\), where

\[
\begin{align*}
G &= \Gamma_1 \div \phi_1 \\
Gs &= [\Gamma_i \div \phi_i]_{i=2}^n \\
Gs' &= [\Gamma_i' \div \phi_i']_{i=1}^m
\end{align*}
\]

By the induction hypothesis we have that

\[
\Gamma_i \vdash \phi_i \text{ for } i \in [2, n] \tag{3.1}
\]

and

\[
\Gamma_i' \vdash \phi_i' \text{ for } i \in [1, m] \tag{3.2}
\]

By Lemma 3.5 and assertion (3.2) we have

\[
\Gamma_1 \vdash \phi_1
\]

The last assertion, together with (3.1), completes the proof of Lemma 3.6 and thus also Theorem 3.3.

Lemma 3.5 states that if we have a proof of all subgoals generated by a tactic, then we can construct a proof of the original goal. Lemma 3.6 is a technical generalization of soundness that allows multiple uses of Lemma 3.5. Note that the proofs in this section are constructive. We will revisit their computational content later, when we add explicit proof terms to the logic.

We now proceed to prove completeness. We again provide a constructive proof, in this case we can extract a procedure that translates derivations in natural deduction into proof scripts. We denote this function using semantic brackets ([\[\Gamma \vdash \phi\]]) and we show it in full in Figure 3.4.\footnote{\texttt{H} in \texttt{assert} is assumed to be fresh.} What remains then is to verify that the result proof script indeed solves the goal.

Proof. By structural induction on the derivation. In case of rules that have more than one premise we use the concatenation lemma (Lemma 2.2).
Figure 3.4: Translation from derivations to scripts
Chapter 4

Proof representation

4.1 Proof assistants and belief

From the user’s point of view interactive theorem proving is like playing a video game: she starts with a single goal and then begins to input commands (in the form of tactics) into the proof assistant. She sees the goal stack grow and shrink, repeatedly hits dead ends and adjusts the assumptions over and over. Yet, hopefully sooner rather than later, her tired eyes witness the blessed **Proof completed** message issued by the proof engine. “I guess the theorem was true after all,” she says, visibly relieved.

Is she correct, though?

In the previous chapters we have formalized both the top level structure of the proof engine and the meaning of a single tactic. We proved that the system is correct with respect to a well-studied framework of natural deduction. In particular, we have shown that if a goal is solved, then there exists a derivation of the goal theorem. So, *would you trust RoCoQo?*

Recall the classic quote from Donald Knuth: *Beware of bugs in the above code; I have only proved it correct, not tried it.* A proof assistant is not a mathematical object, it is often a substantial computer program, consisting of thousands of lines of code\(^1\). The bigger the program, the harder it is to understand it and larger is the chance of bugs (programming mistakes) to be hidden somewhere. Not to mention bugs in the compiler, the run-time system, the operating system and the underlying hardware... As discussed by Pollack in *How to Believe a Machine-Checked Proof* [Pol98], one could ask the machine to construct, behind the scenes, a proof that can be printed in a human-readable form. An expert logician could then verify the output. The problem is that this approach does not scale at all, as computer generated proofs get enormously large very quickly.

\(^1\)According to [ACTZ07] Coq is about 166 000 lines of source code.
Therefore, Pollack argues, a much better way is to have the proof verified by another machine. We only have to make sure (as proof assistant writers) that the output contains all details, including those that have been synthesized by complicated and time-consuming heuristics or have been provided by the user. In such a setting the software that checks the proof (or certificate) can be small and simple enough to be manually inspected and verified by a human. The crux of the argument is that many independent proof checkers can be developed by independent individuals and each time a proof is deemed correct our collective belief in the proof grows.

In practice a proof checker is a part of the proof assistant and it belongs to the so called trusted code base of the prover. A proof assistant is said to satisfy the so-called de Bruijn criterion [BW05] if the correctness of the whole system relies only on the correctness of a very small trusted code base – the kernel.

One could think of many possible ways to design easily checkable proof representations. We could propose many ad-hoc encodings of derivations in natural deduction as trees with some annotations. However, there exists a well understood framework rooted in the studies of logic and lambda calculi: the so-called Curry-Howard Isomorphism [SU06]. We will see the details soon, but the main idea is that proofs are represented as lambda terms (i.e. programs in a minimal functional programming language). Hence proof construction amounts to program synthesis and type inference, while proof checking is reduced to type checking. AUTOMATH, NuPrl, Coq and Matita are examples of provers that use this proofs-as-programs paradigm.

Other ideas for assuring correctness have been proposed. LCF, HOL and Isabelle all use the so-called LCF-approach [GMW79]. Instead of computing explicit proof terms, safety is obtained by providing an abstract type of theorems \texttt{thm} and it is the job of the metalanguage to ensure that those abstraction barriers are preserved. A disadvantage of this approach is that considerable trust must be put in the implementation of the metalanguage. The main selling point is that the values of the \texttt{thm} data type need not be stored in memory. This has been of crucial importance in 1960s and 1970s, because of hardware limitations – for instance, the Edinburgh LCF has been implemented in LISP and had run on a machine with only 256 kilobytes of RAM [GMW79]. Paulson argues that the memory footprint of proofs is still an issue nowadays, because our ambitions have gotten bigger [Pau12].

### 4.2 Typed lambda calculi for proof representation

To explain the ideas behind proof representations based on typed lambda calculi, we review the simplest system of the family: the simply typed lambda calculus (STLC), extended with binary products and sums as well as with the nullary variants.
4.2.1 The simply typed lambda calculus: syntax and semantics

The syntax of types and terms is as follows:

\[ \begin{align*}
\tau & ::= p \mid \text{unit} \mid \text{empty} \mid \tau \ast \tau \mid \tau + \tau \mid \tau \rightarrow \tau \\
\end{align*} \]

\[ \begin{align*}
t & ::= \langle t, t \rangle \mid \text{fst} \ t \mid \text{snd} \ t \\
& \quad \mid 1 \\
& \quad \mid \text{inj}_1 \ t \mid \text{inj}_2 \ t \\
& \quad \mid \text{case} \ t \{ \text{inj}_1 \ x \rightarrow t_1 \mid \text{inj}_2 \ y \rightarrow t_2 \} \\
& \quad \mid \text{abort} \ t \\
& \quad \mid x \mid t \mid \lambda x: \tau. t
\end{align*} \]

The (standard) typing rules are presented in Figure 4.1. The typing context, denoted \( \Gamma \), is a set of (name : type) pairs called type assignments. Each name occurs in \( \Gamma \) at most once, so in rules that extend the context with a new name (\( \text{CASE}, \text{ABS} \)) there is an (implicit) side condition stating that this name is fresh with respect to the context.
It should be noted that we use explicitly typed terms, because our goal is to have the following properties:

1. Whenever $\Gamma \vdash t : \tau$ holds, then $\tau$ is unique.

2. Given $\Gamma$ and $t$ the problem of calculating $\tau$ such that $\Gamma \vdash t : \tau$ should be effectively solvable and the algorithm should be as simple as possible.

We can now implement certificate checking as type inference – we simply calculate the type of the given proof and check that it coincides with the initial formula. The properties above ensure that this algorithm is effective and deterministic.

Only some of the term constructors contain annotations, why is that? The short answer is that in other cases all required type information is available from the subterms (via a recursive call in the type checker). A more principled way is to perform the so called mode analysis, often used in the realm of logic programming (see e.g. lecture notes by Frank Pfenning [Pfe01]). The Twelf implementation of the LF logical framework [PS99] can perform this analysis automatically.

### 4.2.2 Connection with natural deduction

So far we have described the types and the type system of STLC, but our end goal is to obtain a way to represent proofs. The key observation, originally made by Howard in 1969 [How80], is that there is an intimate connection between the natural deduction proof system and STLC. The following table summarizes the dual nature of types and propositions.

<table>
<thead>
<tr>
<th>Natural deduction</th>
<th>STLC</th>
</tr>
</thead>
<tbody>
<tr>
<td>proposition</td>
<td>type</td>
</tr>
<tr>
<td>proof of $\phi$</td>
<td>term of type $\phi$</td>
</tr>
<tr>
<td>provable proposition</td>
<td>inhabited type</td>
</tr>
<tr>
<td>conjunction ($\phi \land \psi$)</td>
<td>product type ($\tau_1 \times \tau_2$)</td>
</tr>
<tr>
<td>disjunction ($\phi \lor \psi$)</td>
<td>sum type ($\tau_1 + \tau_2$)</td>
</tr>
<tr>
<td>implication ($\phi \Rightarrow \psi$)</td>
<td>arrow type ($\tau_1 \rightarrow \tau_2$)</td>
</tr>
<tr>
<td>trivially provable proposition (⊤)</td>
<td>unit type (unit)</td>
</tr>
<tr>
<td>unprovable proposition (⊥)</td>
<td>empty type (empty)</td>
</tr>
</tbody>
</table>

Table 4.1: Curry-Howard Isomorphism for natural deduction and STLC

We should point out that it is not a mere analogy: if we erase the terms from Figure 4.1 we obtain the inference rules of natural deduction for the intuitionistic propositional logic (IPC, Figure 3.1.) Recall that we previously read the judgment $\Gamma \vdash \phi$ as “$\phi$ can be derived from $\Gamma$.” Now we can also explain it using the typing judgment for STLC as “exists a term (here:
proof) \( t \) such that \( \Gamma \vdash t : \tau \). This is why STLC can be used to adequately represent (encode) proofs, provided we chose IPC as our logical framework. This relationship is not accidental as evidenced by a vast amount of literature on the subject. Therefore more expressive logics can be applied in this way, as numerous instances of the proof-as-types paradigm have been invented, discovered and designed ([SU06], [Gri90], [DP01], [DCPT12]). The current version of Coq uses the Calculus of Inductive Constructions [PPM90].

We have thus seen how to represent whole proofs. In the next section we look at atomic tactics and ways of representing partial proofs.

4.3 Tactics as proof builders

4.3.1 Tactics and evidence. Partial proof builders

So far we saw how can one describe tactics as conditional, goal-altering operations. We still need to explain the way tactics support incremental proof construction.

The basic idea has been already laid out: STLC with the typing rules provide a guide. Previously we analyzed the typing judgment for type checking, with inputs \( \Gamma \) and \( t \). This time we will consider \( \Gamma \) and \( \tau \) to be the inputs and the term \( t \) (the proof) to be the output. For example, recall the typing rule for abstraction:

\[
\text{ABS} \quad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x : \tau_1. t : \tau_1 \to \tau_2}
\]

The intro \( x \) tactic takes us from the conclusion to the premise. Now suppose that we have solved the goal represented by the premise and constructed a proof \( p \). Then, the ABS rule says that \( \lambda x : \tau_1. p \) term is a valid proof of the original goal. This is a change of direction: while tactics provide means for backward reasoning, the typing judgments are inductively defined and thus proof construction is inherently top-down.

In order to invert the flow, we need a way to describe partial proofs (in the form of incomplete terms) which we will associate with tactics. There are various possibilities:

1. We can extend the syntax of terms with numbered holes. For the intro example this is \( \lambda x : \tau. \square_1 \),

2. We can extend the terms with a new kind of named variables, called metavariables,

3. We can use a higher-order encoding, and represent holes by functions from a metalanguage, \(^2\)

\(^2\)So in this interpretation \( \Gamma \vdash \phi \) hides (or encapsulates) the proof using an existential type.
4. We can defunctionalize the previous approach and provide an equivalent first-order data type along with an interpretation function.

The first approach is visually attractive, because the proof of the $i$-th subgoal will be substituted for the $i$-th hole. This invariant is nice, but requires care and a lot of index-shifting (which is very error-prone in practice). Matita uses this approach for one of its internal languages [ACTZ07].

The second approach in employed by Kirchner in his adaptation of the $\lambda\mu$ $\tilde{\mu}$ calculus [Kir07]. Compared to the first approach, the proof structure is not obvious at a glance, but it is easy to implement and to reason about.

In the Edinburgh LCF [GMW79] the third approach was used and the tactics had type:

```
type tactic = goal -> (goal list * proof)
and proof = thm list -> thm
```

The proof type was also often called a validation in the LCF literature (including [GMW79]).

Finally, the fourth approach is equivalent to the third one, but would be convenient should be decide to implement a proof assistant in a low level programming language without higher-order functions, for example to use a spare washing machine or freezer for proof mining.

Some proofs take years to complete ([Gon07], [Gon13]), so partial proofs are an important subject have been investigated by several researchers and here we have only reviewed some possible representations. Kirchner provides a more broad review in his PhD thesis [Kir07].

4.3.2 Validations in RoCoQo

Figure 4.2 presents the signature (the abstract interface) for a partial proof representation used in RoCoQo. The `cls_type` OCaml type denotes a closed STLC type.

`builder` denotes the internal representation type and is left abstract in the signature, so that the client code can only manipulate `builders` in a safe, controlled manner (via the other names included in the signature). The `build` function computes the complete proof so it can be only invoked if the partial proof has no gaps remaining. As demonstrated by Pollack [Pol95] and Appel and Felty [AF04], to encode this obligation in the type system would require dependent types [AMM05], so we have to rely on the exception mechanism and to verify the code by hand.

In RoCoQo to provide a structure implementing the signature we use the LCF idea (the third approach), but we also store the arity of the validation for calculations.

The operations in the third group implement typing rules and the comments indicate the rules from Figure 4.1. For example, the validation for
module type BUILDER = sig
  type builder
  val build : builder -> term
  val simple_combine : builder -> builder -> builder
  val semicolon_combine : builder -> builder list -> builder
  val id : builder
  val empty_rec : cls_type -> name -> builder (* abort *)
  val abs : cls_type -> name -> builder (* abs *)
  val make_pair : builder (* prod *)
  val assumption : name -> builder (* var *)
  val one : builder (* unit *)
  val sum : int -> cls_type -> builder (* inj1, inj2 *)
  val app : name -> cls_type list -> builder (* app *)
  val destruct_pair : name -> name * name -> builder (* fst, snd *)
  val destruct_sum : name -> name * name -> builder (* case *)
  val assert_ : name -> cls_type -> builder (* subst lem *)
end

Figure 4.2: Signature for proof builders

intro is implemented as follows

let abs tp1 n = (1, function
  | [ t ] -> Abs (tp1, Open (open_var t n))
  | _   -> failwith "abs_builder: argument error"

An implementation of validations in a vanilla functional programming
language such as OCaml or Haskell 98 cannot avoid partial functions (here failwith is an OCaml function for throwing an ad-hoc exception), but if the
implementation is correct, then the abs function will always be applied to a
singleton list. Moreover, we should as always be careful about variable bind-
ing – for example, the implementation of the Isabelle system has been re-
portedly [Pau90] haunted with variable capture bugs. Here open_var changes
a free variable (represented as a string) into a bound variable (represented
by a de Bruijn index).

Finally, the middle group corresponds to administrative operations on
partial proofs. The simple_combine validation combines proofs from tactics
separated by a dot. Conceptually it means that given partial proofs \( p_1 \) and
\( p_2 \) we insert \( p_2 \) into the first gap in \( p_1 \). The concrete implementation is as
follows:
let simple_combine (n1, f1) (n2, f2) =  
let n = n1 + n2 - 1 in 
let f = fun xs -> 
  let (xs1, xs2) = List.split_n xs n2 in 
  let t1 = f2 xs1 in 
  f1 (t1 :: xs2) 
in (n, f)

This concludes the description of validations for atomic tactics presented in Chapter 3. In Chapter 5 we introduce higher-order tactics, which compose existing tactics. Most of them manipulate goal contexts and do not require any information about validations at all, but there are two exceptions.  

First of all, we have \texttt{idtac} which should by all means be included in the set of first-order tactics. The reason \texttt{idtac} has been historically included among the tacticals is that it is only useful as a placeholder as it literally does nothing:\footnote{In a similar manner ; or \{\} are used in C-like programming languages as an empty statement.}

let id : builder = (1 , function 
  | [ t ] -> t 
  | _ -> failwith "id_builder: argument error"
)

Second of all, we have the tactic composition \texttt{tac} (depending on the variant, denoted either \texttt{t_1; t_2} or \texttt{t_1; ts}). We present its semantics in Chapter 5, but its proof builder takes a partial proof \(p\) with \(n\) gaps and a list \(ps\) of \((\text{exactly})\ n\) proof builders. The result \(p'\) is \(p\) with the \(i\)-th gap replaced with \(i\)-th partial proof. The number of gaps in \(p'\) is the sum of all gaps in \(ps\). The concrete implementation in OCaml is as follows:

let semicolon_combine (n, f) builders = 
let rec iter xs = function 
  | [] -> [] 
  | (n1, f1) :: bs -> 
    let (here, xs') = List.split_n xs n1 in 
    f1 here :: iter xs' bs 
  in 
  let n' = List.fold builders ~init:0 ~f:(fun sum b -> sum + fst b) in 
  (n', function 
    | xs when List.length xs = n' -> 
      f (iter xs builders) 
    | _ -> 
      failwith "semicolon_combine: argument error"
)

The interested reader should again consult the source code of RoCoQo
to analyze the implementation of the remaining builders.

4.3.3 Correctness of atomic tactics revisited

Implementing validations as higher-order functions seems natural, but it may seem that we (or the authors of LCF) have taken them from thin air. Yet it is the exact opposite: we have already seen them in Chapter 3 when we proved the soundness of proof scripts (Lemma 3.5). The proof was constructive and concrete validations can be extracted from it. For example, take a good look at the proof of Lemma 3.6: you should be able to find the simple_combine function – note how we apply a lemma with \((n - 1) + m\) subderivations to obtain the final derivation in the proof.

This precisely is the message of the Curry-Howard Isomorphism – you cannot program if you do not prove, you cannot have types if you do not have a thing for logic.\(^4\)

\(^4\)Another message is that you may have to sacrifice your sanity if you want to use classical logic, but this is a topic for another day.
Chapter 5

LCF tacticals

So far we have studied the basic building blocks of a proof scripts – the atomic tactics. While fundamental, they are somewhat verbose and lead to scripts that are difficult to maintain. The LCF proof assistant [GMW79] introduced tacticals – higher-order tactics, also called tactic combinators. Tacticals enable one to reduce some repetitions in the scripts and also provide a limited form of failure handling.\footnote{In fact, failures in LCF are the direct precursor of the exception mechanism present in modern ML implementations (such as OCaml [Ocs] and Standard ML [AM91]).}

5.1 Semantics of tacticals

Tacticals extend the grammar of atomic tactics to form the category of tactics. We use the following syntax:

\[
t ::= \text{atac \ atomic tactics} \\
| \text{idtac \ identity} \\
| \text{fail \ failure} \\
| t_1 || t_2 \ \text{alternation} \\
| t_1 ; t_2 \ \text{composition} \\
| t_1 ; ts \ \text{branching composition} \\
| T \ \text{tactic variable} \\
| \text{fix} \ T \ t \ \text{fixed-point operator}
\]

\(ts\) denotes a list of tactics. \((t_0 ; ts)\), the branching composition tactical, is written in Coq’s concrete syntax as \(t_0 ; [\ t_1 \ | \ldots \ | \ t_n \ ]\).

Tacticals are called higher-order tactics because they do not modify the goal by themselves, they only provide the “glue” to compose atomic tactics. Therefore, the semantic rules manipulate whole goals only.\footnote{To be precise, idtac and the composition tacticals can be seen as special atomic tactics, because they have to manipulate validations. In contrast, other tacticals need only pass validations around. Details can be found in the RoCoQo implementation.}
The result \( r \) is either \( Gs \), a list of subgoals generated by the tactic \( t \), or \( \bot \), denoting failure. The composition operators require execution in a list of subgoals. For this end we introduce a sequentialization mechanism, which can handle lists of goals and lists of tactics, expressed with the judgment

\[
Gs \triangleright ts \Downarrow \text{seq} r
\]

Note that sequential execution is defined only when \(|Gs| = |ts|\).

Figure 5.1 presents the natural semantics for tactic execution. The semantics extends the rules for atomic tactics (Figures 3.2 and 3.3). The \( n \times t \) syntax used in the \texttt{SEMI-THEN} rule denotes a list comprising of \( n \) copies of \( t \).

We proceed to discuss the semantics of tacticals. \texttt{idtac} never fails and it does nothing (the \texttt{IDTAC} rule). It is the identity tactical and can be used as a placeholder. On the other hand, \texttt{fail} always fails (the \texttt{FAIL} rule). Perhaps surprisingly, the tactical is often used – see the \texttt{solve} idiom in Section 5.2.2.

\( t_1 \parallel t_2 \), the alternative operator, begins by executing \( t_1 \). If \( t_1 \) succeeds (subgoals are generated), then \( t_1 \parallel t_2 \) succeeds as well (rule \texttt{ALT-1}). If \( t_1 \) fails, then \( t_2 \) is executed (rule \texttt{ALT-2}).

The composition \((t_1; t_2)\) and branching composition operators \((t_1; ts)\) both begin by executing \( t_1 \) and both fail if \( t_1 \) fails (rules \texttt{SEMI-FAIL} and \texttt{BRANCH-FAIL-1}, respectively). When \( t_1 \) generates subgoals \( Gs \), then the composition operators use the sequentialization judgment to perform execution in each subgoal. The difference between the operators is that \((t_1; t_2)\) will use \( t_2 \) inside every subgoal (by making \(|Gs| \) copies of \( t_2 \)), while \((t_1; ts)\) executes the \( n \)-th tactic inside the \( n \)-th goal. To make this possible, the branching operator must check that it has been given the right number of tactics (the \(|Gs| = |ts| \) check in rule \texttt{BRANCH-THEN}) and the whole execution fails should the number be wrong (the \texttt{BRANCH-FAIL-2} rule). Sequential execution enters the subgoals from left to right and the whole computation fails as soon as the first failure occurs (rules \texttt{SEQ-FAIL-1} and \texttt{SEQ-FAIL-2}). If all executions succeed, we concatenate the subresults (rule \texttt{SEQ-OK}).

Recursion stops when we run out of goals and tactics at the very same time (rule \texttt{SEQ-DONE}). Finally, we have the fixed-point operator \( \text{fix} \ T t \), used to implement recursive tactics (see example in Section 6.2). To execute such a recursive tactic, we execute the body \( t \) in which all free occurrences of \( T \) have been replaced with the initial expression, thus performing a single unfolding of the fixed point.

\[^3\]We can also say that in the case of \((t_1; t_2)\) it is the proof engine’s job to produce the right number of tactics, while with \((t_1; ts)\) this becomes the responsibility of the user.

\[^4\]An alternative notation, common in the literature, would be \( \mu T.t \)
\[
\begin{align*}
\text{IDTAC} & \quad G \triangleright \text{idtac} \downarrow [G] \\
\text{FAIL} & \quad G \triangleright \text{fail} \downarrow \perp \\
\text{ALT-1} & \quad G \triangleright t_1 \downarrow Gs \\
& \quad G \triangleright (t_1 \parallel t_2) \downarrow Gs \\
\text{ALT-2} & \quad G \triangleright t_1 \downarrow \perp \\
& \quad G \triangleright (t_1 \parallel t_2) \downarrow r \\
\text{SEMI-FAIL} & \quad G \triangleright t_1 \downarrow \perp \\
& \quad G \triangleright (t_1 ; t_2) \downarrow \perp \\
\text{SEMI-THEN} & \quad G \triangleright t_1 \downarrow Gs \\
& \quad G \triangleright Gs \triangleright (|Gs| \times t_2) \downarrow \text{seq} r \\
& \quad G \triangleright (t_1 ; t_2) \downarrow r \\
\text{BRANCH-FAIL-1} & \quad G \triangleright t_1 \downarrow \perp \\
& \quad G \triangleright (t_1 ; ts) \downarrow \perp \\
\text{BRANCH-FAIL-2} & \quad G \triangleright t_1 \downarrow Gs \\
& \quad |Gs| \neq |ts| \\
& \quad G \triangleright (t_1 ; ts) \downarrow \perp \\
\text{BRANCH-THEN} & \quad G \triangleright t_1 \downarrow Gs \\
& \quad |Gs| = |ts| \\
& \quad G \triangleright ts \downarrow \text{seq} r \\
& \quad G \triangleright (t_1 ; ts) \downarrow r \\
\text{SEQ-DONE} & \quad \varepsilon \triangleright \varepsilon \downarrow \text{seq} \varepsilon \\
\text{SEQ-FAIL-1} & \quad G \triangleright t \downarrow \perp \\
& \quad (G : Gs) \triangleright (t : ts) \downarrow \text{seq} \perp \\
\text{SEQ-FAIL-2} & \quad G \triangleright t \downarrow Gs' \\
& \quad Gs \triangleright ts \downarrow \text{seq} \perp \\
& \quad (G : Gs) \triangleright (t : ts) \downarrow \text{seq} \perp \\
\text{SEQ-OK} & \quad G \triangleright t \downarrow Gs' \\
& \quad Gs \triangleright ts \downarrow \text{seq} Gs'' \\
& \quad (G : Gs) \triangleright (t : ts) \downarrow \text{seq} (Gs' \oplus Gs'') \\
\text{FIX} & \quad G \triangleright [T := \text{fix} T t] \downarrow r \\
& \quad G \triangleright \text{fix} T t \downarrow r
\end{align*}
\]

\textbf{Figure 5.1:} Execution of LCF tacticals
5.2 Common idioms and derived tactics

5.2.1 Try

When put on the left, $\text{idtac}$ acts as an annihilator of the alternative operator (i.e. $\text{idtac} \ || \ t$ is equivalent to $\text{idtac}$), but things become more interesting when we put $\text{idtac}$ on the right. It is actually useful to define a new tactical $\text{try} \ t$ as

$$\text{try} \ t := (t \ || \ \text{idtac})$$

with the (derived) semantics

$$\begin{align*}
\text{TRY-OK} & \quad \frac{G \triangleright t \Downarrow Gs}{G \triangleright \text{try} \ t \Downarrow Gs} \\
\text{TRY-FAIL} & \quad \frac{G \triangleright t \Downarrow \bot}{G \triangleright \text{try} \ t \Downarrow [G]}
\end{align*}$$

In short: $\text{try}$ turns a fallible tactic into a non-failing one.

5.2.2 Solve

Often one wants to apply a tactic, but only if it does not generate any subgoals (i.e., it solves the current goal). We can achieve this using the $\text{solve}$ tactical, defined as

$$\text{solve} \ t := (t \ ; \ \text{fail})$$

The derived semantic rules are

$$\begin{align*}
\text{SOLVE-OK} & \quad \frac{G \triangleright t \Downarrow \varepsilon}{G \triangleright \text{solve} \ t \Downarrow \varepsilon} \\
\text{SOLVE-FAIL-1} & \quad \frac{G \triangleright t \Downarrow Gs \quad Gs \neq \varepsilon}{G \triangleright \text{solve} \ t \Downarrow \bot} \\
\text{SOLVE-FAIL-2} & \quad \frac{G \triangleright t \Downarrow \bot}{G \triangleright \text{solve} \ t \Downarrow \bot}
\end{align*}$$

5.2.3 Repeat

Thus far we have seen tactics and tacticals capable of doing only one step of inference at a time. In many cases we want to iterate a tactic until it fails; for instance, most scripts begin with a series of intros. In LCF (and Coq), the repetition tactic is called $\text{repeat} \ t$ and is characterized recursively as

$$\text{repeat} \ t = \text{try} \ (t \ ; \ \text{repeat} \ t)$$

To derive a non-circular definition, we use the fixed-point operator to transform the equation into a non-recursive one, obtaining

$$\text{repeat} \ t := \text{fix} \ T \ (\text{try} \ (t \ ; \ T))$$

However, in Coq $\text{idtac} \ || \ t$ is equivalent to $t$, because of an implicit progress check (explained later).
For this definition we derive the following semantic rules

\[
\begin{align*}
\text{REPEAT-DONE} & : G \triangleright t \Downarrow \bot \\
& \implies G \triangleright \text{repeat } t \Downarrow \{G\} \\
\text{REPEAT-GO} & : G \triangleright t \Downarrow G_s \\
& \implies Gs \triangleright ((Gs \times \text{repeat } t) \Downarrow_{\text{seq}} r) \\
& \implies G \triangleright \text{repeat } t \Downarrow r
\end{align*}
\]

Intuitively, \text{repeat } t executes \( t \) as many times as possible. More precisely, it first executes \( t \) and behaves like identity when \( t \) fails (\text{REPEAT-DONE}). Should \( t \) generate subgoals, then \text{repeat } t executes recursively in every subgoal using the sequentialization judgment (\text{REPEAT-GO}).

It should be noted that \text{repeat } t, as described here, never fails. More importantly, it will loop given a never-failing tactic as argument (e.g., \text{idtac}). As we will see in the next chapter, the latter behavior is taken care of in Coq’s implementation.

Coming back to the motivating example, it should be clear by now that we can introduce all hypotheses at once using \text{repeat intro}. Another case when \text{repeat} proves useful is when the goal has a regular structure, such as \((a \land b \land \top \land c) \land \top \land (d \land \top)\). The tactic \text{repeat split ; (try truth)} will decompose all top-level conjunctions and (whenever possible) solve all instances of the trivial goal \( \top \), thus leaving the user with 4 subgoals: \( a, b, c, d \).

### 5.2.4 Branching composition

Finally, it is instructive to see the branching composition operator in action. One common use case is when proofs for each of the subgoals are similar, but, e.g., the first tactics differ. For example, the script:

<table>
<thead>
<tr>
<th>\text{Goal } a+b \rightarrow b+a.</th>
</tr>
</thead>
<tbody>
<tr>
<td>intro H.</td>
</tr>
<tr>
<td>destruct H.</td>
</tr>
<tr>
<td>right. trivial.</td>
</tr>
<tr>
<td>left. trivial.</td>
</tr>
</tbody>
</table>

can be factored into a single tactic as

<table>
<thead>
<tr>
<th>Goal a+b -&gt; b+a.</th>
</tr>
</thead>
<tbody>
<tr>
<td>intro H; destruct H; [ right</td>
</tr>
</tbody>
</table>

If we need to “do nothing” in some of the branches, then \text{idtac} can be used as a placeholder.
5.3 The composition operator vs. the dot command

It is interesting to compare the semantics of the composition operator (in the form of the extended judgment) with the semantics of the dot command (introduced in Chapter 2):

\[
\begin{align*}
\text{TAC-OK} & : \quad G \triangleright e \Downarrow Gs' \\
& \Rightarrow (G : Gs) \triangleright (e : es) \rightarrow (Gs' \oplus Gs) \triangleright es \\
\text{SEQ-OK} & : \quad G \triangleright t \Downarrow Gs' \\
& \Rightarrow (G : Gs) \triangleright (t : ts) \Downarrow (Gs' \oplus Gs'') \\
\end{align*}
\]

The dot operator (TAC-OK) is very local and works in a last-in first-out fashion, while composition proceeds in parallel and each subgoal is solved independently (SEQ-OK).

5.4 Summary

We have given a formal semantics for many classical tactic combinators, including the five tacticals presented in the description of the Edinburgh LCF [GMW79].

We should note that we have only begun the journey towards robust proof scripts: judicious use of tacticals can lead to improvements in the structure of the proof scripts, but overusing them can make the script even more feeble than before. The reason is that simple tacticals can trigger automatic name generation (e.g. repeat intro), but there are very few ways to refer to an assumption without using its name – assumption is the only exception. The match goal construct of Ltac, which we describe in the next chapter, fills this void nicely.
Part III

Operational Foundation for Ltac
Chapter 6

Ltac – the tactics

In this chapter we begin to describe the Ltac tactic programming language in its contemporary form. In the first step we revisit LCF tacticals from the previous chapter and we introduce the goal matching construction, arguably the killer feature of Ltac. The set of tacticals as described in this chapter faithfully represents the situation in modern Coq and in many cases the semantics presented here will suffice to clarify the details of tactic execution. However, should one want to use the programming language aspects of Ltac, then the extended semantics from Chapter 7 should be consulted instead.

6.1 The semantics of failure based backtracking

We extend and update the grammar of LCF tactics to obtain (a first approximation of) Coq tactics. The syntax is the following

\[
\begin{align*}
t & ::= \text{atac} & \text{atomic tactics} \\
 & | \text{idtac} & \text{identity} \\
 & | \text{fail } n & \text{failure on level } n \\
 & | \text{first } ts & \text{generalized alternation} \\
 & | t_1 ; t_2 & \text{composition} \\
 & | t_1 ; ts & \text{branching composition} \\
 & | T & \text{tactic variable} \\
 & | \text{fix } T t & \text{fixed-point operator} \\
 & | \text{progress } t & \text{progress check} \\
 & | \text{mgoal } cls & \text{goal context pattern matching}
\end{align*}
\]

\(ts\) denotes a list of tactics as before, while \(cls\) denotes a list of clauses. A clause \(cl\) consists of a goal context pattern \(p\) and a tactic. We do not specify the syntax of patterns here – see examples later in the section. Following Coq, we generalize the \(t_1 || t_2\) operator to \textbf{first} \(ts\).\(^1\) In a similar spirit, \textbf{fail} is

\(^1\)\textbf{first} \([t_1 | \ldots | t_n]\) is equivalent to \(t_1 || \ldots || t_n\).
now given a level argument n. \texttt{mgoal} is an abbreviation for the \texttt{match goal} construction.

Coq inherits the tacticals from LCF, its predecessor, and at a glance we have simply added two new constructions to the grammar. However, there are important changes in the underlying semantics. To explain the execution of Ltac’s tacticals, we redefine the judgment

\[ G \triangleright t \Downarrow r \]

As before, \( r \) can be a list of subgoals generated by \( t \) or an error, but this time failure is denoted as \( \bot_n \), with \( n \) being the level marker. To explain the concept of levels we need to introduce some terminology. We say that a tactical is a failure handler if it traps failures rather than to simply propagate them. Hence \( t_1 || t_2 \) is the sole failure handler among primitive LCF tacticals.\(^2\)

The semantics of LCF tacticals from the previous chapter is generalized as follows: failure handlers treat \( \bot_0 \) as \( \bot \) and they decrement the level of positive level failures \( \bot_{s(n)} \), while all other tacticals treat \( \bot_n \) as \( \bot \). So the biggest change is that by prescribing a failure with a sufficiently high level (via \texttt{fail n}) we can bypass any (fixed at call site) number of failure handlers. This is much in contrast to the previous situation, in which \texttt{try t} would encapsulate any failure arising during the execution of \( t \) – now we might have to stack a few layers of the \texttt{try} tactical. Unfortunately, there does not seem to be a way to capture failure at any level (if such a tactical were to exist, \texttt{fail \infty} would be a suitable name). This looks unwieldy in theory, but in practice does not seem to be a problem. Moreover, the new behavior creates opportunities for interesting programming idioms to emerge – see Section 7.3.1 for an example.

Figure 6.1 presents the updated rules for tacticals, while Figure 6.2 contains rules for the new \texttt{match goal} construction. We postpone the introduction of \texttt{progress} until Section 6.2.2 as it is useful only in special cases.

\subsection{Semantics of \texttt{match goal}}

We now proceed to explain and discuss one of the most interesting features of Ltac as a whole, the goal matching construction. First of all, \texttt{match goal} is based on the popular pattern matching feature known from functional programming languages, but instead of scrutinizing expressions, we can pattern match on (the shape of the) goal formula and (even at the same time) on the types of multiple assumptions present in the goal context.

In Coq’s concrete syntax this is written roughly as

\(^2\)Tactics that are derived from the alternation tactical, e.g. \texttt{try t} and \texttt{repeat t}, become failure handlers themselves.
Figure 6.1: Tactic execution – tacticals
We can match an arbitrary number of assumptions (including none), but if we do not want to match the goal formula, we have to use _, the wildcard pattern. Consider the following example tactic:

```
match goal with
  | [ |- unit ] => truth
  | [ H : absurd |- _ ] => exfalso; apply H
end
```

We should note that when we match an assumption (patterns to the left of |-), we provide a local name which can be used by the tactic on the right hand side. We can also select assumptions based on their type, and the pattern variables (beginning with ? when used on the left hand side and without it on the right) can be shared between hypotheses and the goal formula. One crucial question emerges: how does the proof engine select assumptions when matching? An inherent property of the vanilla pattern matching of ML is that it is entirely deterministic. In contrast, the `match goal` construction introduces a kind of non-deterministic pattern instantiation search. Precisely, the documentation of the 8.4 version of Coq [Tea13] states that “hypothesis patterns are examined from right to left (...). For each hypothesis pattern, the goal hypotheses are matched in order (fresher hypothesis first).” When we find a combination of hypotheses that satisfies the whole pattern of a given clause, we instantiate the variables and execute the tactic on the right. Should this execution fail, we look for another matching combination, and only after we exhaust all possibilities, do we begin to analyze the next clause. When does this madness culminate? When a tactic on the right hand side generates new subgoals.

For practical examples of the uses of `match goal` see Sections 6.3 and 6.4.

We hope that the reader is now under the impression that the instantiation search can be rather technical and awkward to state formally, let alone to be presented among the rules for the `match goal` tactical. We therefore introduce an abstraction of a matcher $m$ equipped with two operations:

1. `start G p` which initializes the instantiation search for pattern $p$ inside goal $G$,

2. `next m` which returns `Done` when all possibilities have been generated and otherwise returns `Match(σ, m')`, where $σ$ is a functional represen-
tation of the substitution of hypotheses’ names and types and \( m' \) is the matcher with updated state.

In short, a matcher is a purely functional instance of the iterator pattern [GOF04]. A natural implementation can utilize lists (or better: lazy lists). It is also possible to use the two-continuation model of backtracking, by the help of the LogicT library [KSFS05]. An interested reader should refer to the source files of RoCoQo for a complete example. Finally, we should note that we also abstract over the concrete grammar of patterns, so that we can provide a modular account of the essentials of match goal.

We can finally move on to analyze the formal rules in Figure 6.2. The matching fails when the list of clauses is empty (the \texttt{mgoal-nil} rule). Otherwise, we take the first clause and instantiate the matcher abstraction (rule \texttt{mgoal-cons}). Since we have to iterate the instantiation generation, we use a helper judgment of the form

\[ G \vdash m \cdot t \cdot \text{cls} \Downarrow \text{pat} \]

To analyze a clause, we first apply the next function. If we are done, then the execution of the current clause fails and we move to the next one (rule \texttt{pat-done}). Otherwise, we perform substitution on the right hand side tactic and execute it. If this fails with level zero, then we use the updated matcher and try another instantiation (rule \texttt{pat-next}), but when the level is positive we abort the whole computation and decrement the level (rule \texttt{pat-jump}). Finally, if \( \sigma(t) \) succeeds then matching is finished (rule \texttt{pat-or}).

### 6.2 Derived tactics revisited

#### 6.2.1 Alternation and failure trapping

\texttt{first} can be used to define the tacticals that we have seen in the previous chapter:\(^3\)

\[
\text{first } (t_1 | | t_2) := \text{first } [t_1 | t_2 ] \\
\text{try } t := \text{first } [ t | \text{idtac} ]
\]

The derived rules for the tactics are the following:

\[\begin{align*}
\text{ALT-OK: } & G \vdash t_1 \Downarrow Gs \\
& \Rightarrow G \vdash (t_1 | | t_2) \Downarrow Gs
\end{align*}\]

\[\begin{align*}
\text{TRY-OK: } & G \vdash t \Downarrow Gs \\
& \Rightarrow G \vdash \text{try } t \Downarrow Gs
\end{align*}\]

\[\begin{align*}
\text{ALT-FAIL-1: } & G \vdash t_1 \Downarrow \bot_0 \\
& \Rightarrow G \vdash (t_1 | | t_2) \Downarrow r
\end{align*}\]

\[\begin{align*}
\text{TRY-FAIL-1: } & G \vdash t \Downarrow \bot_0 \\
& \Rightarrow G \vdash \text{try } t \Downarrow [G]
\end{align*}\]

\[\begin{align*}
\text{ALT-FAIL-2: } & G \vdash t_1 \Downarrow \bot_{s(n)} \\
& \Rightarrow G \vdash (t_1 | | t_2) \Downarrow \bot_n
\end{align*}\]

\[\begin{align*}
\text{TRY-FAIL-2: } & G \vdash t \Downarrow \bot_{s(n)} \\
& \Rightarrow G \vdash \text{try } t \Downarrow \bot_n
\end{align*}\]

\(^3\)To be precise, in Coq we have rather \( (t_1 | | t_2) := \text{first } [ \text{progress } t_1 | t_2 ] \) (see next section.)
50 CHAPTER 6. LTAC – THE TACTICS

\[
\begin{align*}
\text{MGOAL-NIL} & \quad G \vdash \text{mgoal} \varepsilon \Downarrow_0 \\
\text{MGOAL-CONS} & \quad G \vdash \text{start} \ G \ p \cdot t \cdot \text{cls} \Downarrow_{\text{pat}} r \\
& \quad G \vdash \text{mgoal} \ ((p, t) : \text{cls}) \Downarrow r \\
\text{PAT-DONE} & \quad \text{next } m = \text{Done} \ G \vdash \text{mgoal} \ \text{cls} \Downarrow r \\
& \quad G \vdash m \cdot t \cdot \text{cls} \Downarrow_{\text{pat}} r \\
\text{PAT-NEXT} & \quad \text{next } m = \text{Match}(\sigma, m') \ G \vdash \sigma(t) \Downarrow_0 \ G \vdash m' \cdot t \cdot \text{cls} \Downarrow_{\text{pat}} r \\
& \quad G \vdash m \cdot t \cdot \text{cls} \Downarrow_{\text{pat}} r \\
\text{PAT-JUMP} & \quad \text{next } m = \text{Match}(\sigma, m') \ G \vdash \sigma(t) \Downarrow_{\sigma(n)} \\
& \quad G \vdash m \cdot t \cdot \text{cls} \Downarrow_{\text{pat}} \bot_n \\
\text{PAT-OK} & \quad \text{next } m = \text{Match}(\sigma, m') \ G \vdash \sigma(t) \Downarrow Gs \\
& \quad G \vdash m \cdot t \cdot \text{cls} \Downarrow_{\text{pat}} Gs
\end{align*}
\]

Figure 6.2: Tactic execution – match goal

\[
\begin{align*}
\text{PROGRESS-FAIL-1} & \quad G \vdash t \Downarrow_n \ G \vdash \text{progress} \ t \Downarrow \bot_n \\
\text{PROGRESS-FAIL-2} & \quad G \vdash t \Downarrow [G] \ G \vdash \text{progress} \ t \Downarrow \bot_0 \\
\text{PROGRESS-OK} & \quad G \vdash t \Downarrow Gs \ Gs \neq [G] \ G \vdash \text{progress} \ t \Downarrow Gs
\end{align*}
\]

Figure 6.3: Tactic execution – progress

Compared to the previous semantics, this time we had to add a special case for \( \bot_{\sigma(n)} \).

6.2.2 Updated repeat

Compared to the LCF version of \texttt{repeat} shown in Section 5.2.3, modern versions of Coq use a slightly improved definition, namely

\[
\text{repeat } t := \text{fix } T \ (\text{try } (\text{progress } t \ ; \ T))
\]

The semantics of the \texttt{progress} tactical used above are presented in Figure 6.3. \texttt{progress } \( t \) fails when execution of \( t \) does not modify the goal at all, i.e., when \( t \) turns out to be equivalent to \texttt{idtac} (rule \texttt{PROGRESS-FAIL-2}). Otherwise,
the result (be it a failure or success) is propagated (rules \texttt{progress-fail-1} and \texttt{progress-ok}, respectively).

Therefore, by wrapping the argument of \texttt{repeat} with an implicit \texttt{progress}, we prevent infinite looping of tactics such as \texttt{repeat idtac}.

Given this updated definition of \texttt{repeat} we can derive the following rules

\[
\text{REPEAT-DONE} \quad \frac{G \triangleright \text{progress } t \downarrow \bot_0}{G \triangleright \text{repeat } t \downarrow [G]}
\]

\[
\text{REPEAT-FAIL} \quad \frac{G \triangleright \text{progress } t \downarrow \bot_{s(n)}}{G \triangleright \text{repeat } t \downarrow \bot_n}
\]

\[
\text{REPEAT-GO} \quad \frac{G \triangleright \text{progress } t \downarrow Gs \quad Gs \triangleright (|Gs| \times \text{repeat } t) \downarrow r}{G \triangleright \text{repeat } t \downarrow r}
\]

Once again, we have a new rule for the \(\bot_{s(n)}\) case.

\section{Small examples}

\subsection{Reimplementation of assumption}

The \texttt{match goal} construction is very useful for building scripts that are resilient against small changes of the proof context (such as renaming of the hypotheses). This is because, among other things, we can refer to the assumptions using local names. As an example, we implement our own version of \texttt{assumption}. Consider the following tactic

\begin{verbatim}
match goal with
  | [ H : _ |- _ ] => solve (apply H)
end
\end{verbatim}

The instantiation engine will select all hypotheses from the context in turn, until the tactic on the right hand side succeeds or we run out of assumptions to try (then the whole \texttt{match goal} will fail). We can also try an approach that utilizes non-linear patterns.

\begin{verbatim}
match goal with
end
\end{verbatim}

This version is much more direct and (possibly) efficient: by using a non-linear goal pattern we no longer walk in the dark – we select exactly the assumption we need. If such an assumption is found, then we \texttt{know} that \texttt{apply} will finish the proof, so we need not use the \texttt{solve} idiom anymore.

\footnote{Of course, infinite computations are still possible, because we allow unbounded recursion.}
6.3.2 Goal context cleanup

When we are in the middle of a proof that deals with many conjunctions and disjunctions it is possible that, after repeated elimination of the connectives, the list of assumptions becomes very long and it becomes difficult to figure out the next steps. Coq comes equipped with the `clear H` tactic, which removes $H$ from the context. It would be somewhat cumbersome to use it by hand – again, proof context matching can prove very handy for automating this kind of task. Consider the following tactic

```coq
repeat
  match goal with
    | [ H : unit |- _ ] => clear H
  end
```

The first clause finds two (different) assumed proofs of the same formula and removes one of them, while the second clause removes all proofs of $\top$ – it has no elimination rules, so we could not use those assumptions anyway. The `repeat` on the outside will invoke the simplification as many times as possible.

6.4 Extended example: propositional reasoning

Consider the following script

```coq
Ltac tauto :=
  repeat intro; try (assumption || truth);
  match goal with
    | [ ?H : _ * _ |- _ ] => destruct H; clear H; tauto
    | [ ?H : _ + _ |- _ ] => destruct H; clear H; tauto
    | [ ?H : _ -> _ |- _ ] => solve (apply H; tauto)
    | [ |- _ * _ ] => split; tauto
    | [ |- _ + _ ] => first [ solve (left; tauto) | solve (right; tauto) ]
    | _ => idtac
  end.
```

The `tauto` procedure, while incomplete, is still pretty powerful, especially considering that it is only a few lines of code. It is also a good example of the heuristic nature of decision procedures written in tactical languages: in the first line we introduce any outstanding implications (`repeat intro`) and then we try to finish the proof. The remaining part demonstrates the versatility of the backtracking semantics of `match goal`. In the first three clauses of the `match goal` we look at the shape of the assumptions and proceed accordingly: we break conjunctions and disjunctions into subcomponents (the use of `clear`
prevents the tactic from looping) and we try to apply any implication. This application is unsafe in the sense that it is possible to make a wrong choice: for instance, if both empty → goal and unit → goal are available, then picking the former might get the tactic stuck. We therefore use the solve idiom – if tauto cannot solve the goal using the implication chosen by the pattern matching procedure, then we backtrack and try a different one. If the goal is a conjunction (fourth clause) then we split it and call tauto recursively. If the goal is a disjunction then we have to make a guess: we first pick left and try to finish the proof. If we fail, the proof state is backtracked and we pick right. The final clause match goal always matches so tauto never fails and the user can try to finish the remaining parts of the proof.

Here are some example goals that tauto is able to solve:

Goal (a->b->c->d->e->f->a).
Goal ((a->b)->a->b).
Goal ((a->b)->(b->c)->(c->d)->a->d).
Goal unit.
Goal (empty -> empty).
Goal (a*b -> b*a).
Goal (a*b*c*d -> d*c*b*a).
Goal (a+b -> b+a).
Goal (a+b+c+d -> d+c+b+a).
Goal ((a+b)+c -> a+(b+c)).
Goal ((a*b->c) -> a->b->c).
Goal ((a->b->c)->a*b->c).
Goal ((a+b->c) -> (a->c)*(b->c)).
Goal ((a->c)*(b->c) -> a+b -> c).

6.5 Summary

We have provided a formal description of the contemporary version of the goal matching construction of Ltac and of the error propagation mechanism present in modern versions of Coq. While the semantics should be already applicable for day-to-day theorem proving, it does not show the whole story. For example, we lack a proper argument passing machinery. Instead of an ad-hoc solution, Ltac is built upon an untyped functional programming language, which opens up a plethora of exciting possibilities. However, a marriage of languages is doomed to introduce subtle interactions and surprising corner cases. It is a Pandora’s box and we attempt to open it wide in the next chapter.
Chapter 7

Ltac – the expressions

In this chapter we finally describe the true nature of the tactic language of Coq. In Chapters 3, 5 and 6 we have focused on the atomic tactics, tacticals and tactic execution. After all, this is what interactive theorem proving is all about. However, Ltac is more than that: it is also an untyped functional programming language and we want to describe and study it as such. From the user’s point of view, the inclusion of language constructs such as function abstraction and function application enables creation of custom tacticals. From our point of view it creates a tension between tactics and expressions, which is exemplified by introducing modes of computation in the semantics presented in this chapter. In consequence, the main judgment is decomposed into expression evaluation and tactic execution.

7.1 Syntax of CoreLtac

We introduce 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. 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:

\[
\begin{align*}
  e \ ::= & \quad v \quad \text{value} \\
             & \quad x \quad \text{variable} \\
             & \quad e \ es \quad \text{application} \\
             & \quad \text{let } x := e_1 \text{ in } e_2 \quad \text{local binding} \\
             & \quad \text{mgoal } \vec{c} \quad \text{match goal} \\
             & \quad \text{fix } x e \quad \text{fixed-point operator}
\end{align*}
\]
\[\begin{align*}
es &::= \varepsilon \mid (e : es) \\
c\ell &::= (p, e) \\
v &::= \lambda \vec{x}.e \quad \text{abstraction} \\
&\quad \mid t \quad \text{tactic} \\
&\quad \mid \lceil n \rceil \quad \text{Ltac’s integer} \\
a &::= x \mid \lceil n \rceil \\
t &::= \textit{atac} \quad \text{atomic tactic} \\
&\quad \mid \textit{idtac} \quad \text{identity} \\
&\quad \mid \textit{fail} a \quad \text{failure} \\
&\quad \mid e_1 ; e_2 \quad \text{composition} \\
&\quad \mid e ; es \quad \text{branching composition} \\
&\quad \mid \textit{first} es \quad \text{generalized alternation} \\
&\quad \mid \textit{progress} e \quad \text{progress check}
\end{align*}\]

We have introduced the syntactic category of expressions \(e\) which is defined mutually with the category \(t\) of tactics. In comparison with the tactics from previous chapters, the syntax is generalized and the arguments of tacticals can now be not only tactics, but arbitrary expressions.

The grammar of expressions includes typical constructs available in functional programming languages: variables, (multi-argument) lambda abstraction and (generalized) function application. In Ltac the syntax for the user requires a variable in the function position of all applications (so \texttt{let} is required when, e.g., the function needs to be computed from nested application), but we relax this restriction for technical reasons (we use substitution in the semantics). We must point out that in Ltac \texttt{match goal} is an expression rather than a tactic, as this design choice has many consequences in the semantics and it may lead to some surprising interactions.

Values include tactics, natural numbers, and lambda abstractions. In CoreLtac natural numbers are of limited use – they can only be used as arguments of the \texttt{fail} tactical. However, they could also be used by atomic tactics: for example, in Coq the \texttt{auto} automation tactic can be given an argument denoting the maximal depth of the proof search.

As in the previous chapter, we do not commit to any particular grammar of proof context patterns (denoted \(p\)).
Figure 7.1: Natural semantics – expression execution

Figure 7.2: Natural semantics – extended expression evaluation
\[
\begin{align*}
\text{IDTAC} & \quad G \triangleright \text{idtac} \downarrow x [G] \\
\text{FAIL} & \quad G \triangleright \text{fail} \downarrow n \downarrow n \\
\text{FIRST}_1 & \quad G \triangleright \text{first} \ \varepsilon \downarrow x \downarrow 0 \\
\text{FIRST}_2 & \quad G \triangleright e \downarrow Gs \\
\text{FIRST}_3 & \quad G \triangleright \text{first} \ (e : es) \downarrow x Gs \\
\text{FIRST}_4 & \quad G \triangleright e \downarrow 0 \\
\text{PROGR}_1 & \quad G \triangleright \text{progress} \ e \downarrow x \downarrow n \\
\text{PROGR}_2 & \quad G \triangleright e \downarrow [G] \\
\text{PROGR}_3 & \quad G \triangleright e \downarrow Gs \quad Gs \neq [G] \\
\text{SEMI}_1 & \quad G \triangleright e_1 \downarrow n \\
\text{SEMI}_2 & \quad G \triangleright e_1 \downarrow Gs \quad Gs \triangleright (|Gs| \times e_2) \downarrow x r_x \\
\text{BRANCH}_1 & \quad G \triangleright e_1 \downarrow n \\
\text{BRANCH}_2 & \quad G \triangleright e_1 \downarrow Gs \quad |Gs| \neq |es| \\
\text{BRANCH}_3 & \quad G \triangleright e_1 \downarrow Gs \quad |Gs| = |es| \\
\text{SEQ}_1 & \quad \varepsilon \triangleright \varepsilon \downarrow x r_x \\
\text{SEQ}_2 & \quad G \triangleright e \downarrow \downarrow n \\
\text{SEQ}_3 & \quad G \triangleright e \downarrow Gs' \\
\text{SEQ}_4 & \quad G \triangleright e \downarrow Gs' \\
\end{align*}
\]

Figure 7.3: Natural semantics – tactic execution
\[
\begin{align*}
\text{VAL} & \quad G \vdash v \mid_v v \\
\text{APP}_1 & \quad \frac{G \vdash e \mid_v \perp_n}{G \vdash e \ es \mid_v \perp_n} \\
\text{APP}_3 & \quad \frac{G \vdash e \mid_v \lambda \vec{x}.e}{G \vdash es \mid_{\text{args}} \perp_n} \\
\text{APP}_2 & \quad \frac{G \vdash e \mid_v \lambda \vec{x}.e}{G \vdash es \mid_{\text{args}} \vec{v}} \quad |\vec{v}| = |\vec{x}| \\
\text{ARGS}_1 & \quad \frac{G \vdash \varepsilon \mid_{\text{args}} \varepsilon}{G \vdash (e : es) \mid_{\text{args}} \perp_n} \\
\text{ARGS}_3 & \quad \frac{G \vdash e \mid_v \perp_n}{G \vdash (e : es) \mid_{\text{args}} \perp_n} \\
\text{ARGS}_5 & \quad \frac{G \vdash es \mid_{\text{args}} \vec{v}}{G \vdash (e : es) \mid_{\text{args}} \vec{v}} \\
\text{LET}_2 & \quad \frac{G \vdash e_1 \mid_v \perp_n}{G \vdash \text{let } x := e_1 \text{ in } e_2 \mid_v \perp_n} \\
\text{MG}_1 & \quad \frac{G \vdash \text{mgoal } \varepsilon \mid_v \perp_0}{G \vdash \text{start } G \ p \cdot e \cdot \vec{c} \mid_v \text{pattern } r_{\text{ex}}} \\
\text{PAT}_1 & \quad \frac{\text{next } m = \text{Done}}{G \vdash \text{mgoal } \vec{c} \mid_v \text{pattern } r_{\text{ex}}} \\
\text{PAT}_3 & \quad \frac{\text{next } m = \text{Match}(\sigma, m')}{G \vdash \sigma(e) \mid_{\text{ex}} s(n)} \\
\text{MG}_2 & \quad \frac{G \vdash \text{mgoal } \vec{c} \mid_v \perp_0}{G \vdash \text{start } G \ p \cdot e \cdot \vec{c} \mid_v \text{pattern } r_{\text{ex}}} \\
\text{PAT}_2 & \quad \frac{\text{next } m = \text{Match}(\sigma, m')}{G \vdash \sigma(e) \mid_{\text{ex}} \perp_0} \\
\text{PAT}_4 & \quad \frac{G \vdash m \cdot e \cdot \vec{c} \mid_{\text{pattern}} \perp_n}{G \vdash m \cdot e \cdot \vec{c} \mid_{\text{pattern}} \perp_n}
\end{align*}
\]

Figure 7.4: Natural semantics – expression evaluation
7.2 Natural Semantics

In this section we introduce a natural semantics for 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 [Tea13].

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. On the other hand, the notion of failures is common to both parts of the language. So far we have seen that a tactic can fail because its precondition does not hold. Moreover, 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. Now that we have moved from a simple tactic calculus to an untyped programming language we have also to account for dynamic type errors.

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 7.3 and 7.4. 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. Some of the judgments return only a limited subset of all possible results. The following table can be used for reference:

<table>
<thead>
<tr>
<th>result</th>
<th>possible forms</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_v )</td>
<td>( ⊥ )</td>
</tr>
<tr>
<td>( r_{vx} )</td>
<td>( ⊥ )</td>
</tr>
<tr>
<td>( r_x )</td>
<td>( ⊥ )</td>
</tr>
<tr>
<td>( r_{args} )</td>
<td>( ⊥ )</td>
</tr>
</tbody>
</table>

As before, the \( ⊕ \) operator denotes concatenation of goal lists, used to flatten lists of subgoals into a single subgoal list. \([G]\) denotes the singleton list containing \( G \).
7.2.1 The notion of modes

Consider the following (artificial) Ltac script:

```cpp
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:

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

we should treat `e1` and `e2` differently: while both `e1` and `e2` should be evaluated, only `e2` 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.

7.2.2 The main judgments

The first judgment denotes tactic execution and is written

$$G \vdash t \downarrow_x r_x$$

The result can be either a list of subgoals `Gs` or a level-`n` failure `⊥_n`, the semantic representation of `fail` `⌈n⌉`. This judgment is based on the tactic execution relation from the previous chapter, but since the arguments of tacticals are now expressions rather than tactics, we need to evaluate the arguments into tactics before we can perform execution. The rules for tactic execution are shown in Figure 7.3.

The second judgment formalizes expression evaluation and is written

$$G \vdash e \downarrow_v r_v$$

The result can be a value `v`, a list of subgoals `Gs` or a level-`n` failure `⊥_n`. It is somewhat surprising that we include `Gs` in `r_v` – this is caused by the fact that `match goal` is an expression rather than a tactic. The rules for expression evaluation are shown in Figure 7.4.

7.2.3 Composition of evaluation and execution

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

$$G \vdash e \downarrow_x r_x$$

This judgment can be summarized as
1. Evaluate $e$ to tactic $t$

2. Execute $t$

Formal rules (given in Figure 7.1) 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.

Whenever a top-level tactic execution is requested (in Coq, this is the dot command, denoted $e.$), 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

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

The rules are presented in Figure 7.2. 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 $\bot_n$.

### 7.2.4 Other judgments

As before, we have helper judgments for sequential tactic execution:

$$Gs \triangleright e \downarrow_{seq} r_x$$

and for pattern instantiation backtracking

$$G \triangleright m \cdot e \cdot \vec{cl} \downarrow_{pattern} r_{vx}$$

Just as in Chapter 6 we abstract over pattern matching and the particular strategy for pattern instantiation backtracking and the judgment is just as before, with the only change being the update from plain tactic execution to extended expression evaluation, as in Ltac the right hand side of each clause can be an arbitrary expression.

We also introduce the judgment:

$$G \triangleright es \downarrow_{args} r_{args}$$

for sequential argument evaluation, because in Ltac is strict and all arguments need to be evaluated before the function is applied.
7.2.5 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, evaluation of the matching construction can return a list of subgoals, thus (by induction) any evaluation can return a list of subgoals. This behavior is very counter-intuitive, but this is the price we have to pay for including match goal in the category of expressions.

On the other hand, to evaluate let \( x := e_1 \) in \( e_2 \) we first evaluate \( e_1 \) to a result \( r \), which 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

\[
\text{let } x := \text{match goal with} \\
\quad | _ => \text{eauto} \\
\text{end} \\
in \text{idtac } x.
\]

Coq produces the following error message:\footnote{This may take a moment, as eauto can lead to a long proof search.}

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 \cite{Chaudron} is to coerce the match goal expression into a tactic by replacing it with

\[
(\text{idtac } ; \text{match goal with } \ldots \text{ end})
\]

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 \cite{Tea13}.

Figure 7.5 presents the semantics of the lazymatch goal construction. The rules are based on the corresponding rules for match goal, the only difference is that rules LPAT2, LPAT3 and LPAT4 only evaluate the expression on the right-hand side, while before we have used the extended evaluation judgment. The consequence is that when the right-hand side expression evaluates to
a tactic \( t \), then in case of \texttt{lazymatch} \( t \) is returned as the result, while \texttt{match} \( t \) executes \( t \) on the spot. Therefore, as a rule of thumb, one should always prefer

\begin{verbatim}
let x := lazymatch goal with ...
\end{verbatim}

over

\begin{verbatim}
let x := match goal with ...
\end{verbatim}

Most of the time they are equivalent, but the second version is bound to mysteriously fail in some cases (described above), so the first variant is the idiomatic usage.
7.3 Examples

7.3.1 Simple objective, complex tactic

The \texttt{tauto} decision procedure does not include any forward chaining for implication, that is, we did not include a clause such as

\[
\begin{array}{l}
| [ H : ?A \rightarrow ?B, H1 : ?A |- _ ] \Rightarrow \text{assert } H2 \ B; \ [ \text{apply } H ; \text{apply } \ H1 \mid \text{idtac} ] \\
\end{array}
\]

The reason is our procedure would loop, while it is not clear which of hypotheses \((H\) or \(H1\)) should be removed after a reasoning step. We would like to be able to have a way to check if a given assumption is already present in the goal context. For this, consider the following Ltac tactic, taken from the Coq textbook by Chlipala [Chl]:

\[
\begin{array}{l}
\text{Ltac } \text{notHyp } P := \\
\quad \text{match goal with} \\
\quad \quad | [ _ : P |- _ ] \Rightarrow \text{fail 1} \\
\quad \quad | _ \Rightarrow \\
\quad \quad \quad \text{match } P \text{ with} \\
\quad \quad \quad \quad | ?P1 \land \ ?P2 \Rightarrow \\
\quad \quad \quad \quad \quad \text{first} \ [ \text{notHyp } P1 \mid \text{notHyp } P2 \mid \text{fail 2} ] \\
\quad \quad \quad \quad | _ \Rightarrow \\
\quad \quad \quad \quad \quad \text{idtac} \\
\quad \end{array}
\]

The intention is that \texttt{notHyp} should succeed whenever there exists a subformula of \(P\) that is not among the assumptions of the proof context. Chlipala uses \texttt{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, \texttt{first} and \texttt{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 \texttt{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.
7.4 Summary

In this chapter we have presented a natural semantics for an interesting subset of Ltac. We believe that this work will fill a gap in the literature on Coq and its tactical language. We now proceed to review the literature on formal semantics of tactics, tacticals and tactic programming languages. We focus on the articles concerning Coq and we discuss the differences with our contributions.

The original description of the tactics and tacticals of the LCF system by Gordon et al. [GMW79] explained atomic tactics semi-formally as transformation of goals into subgoals while for the (some of the less complicated) tacticals the ML code was presented. The semantics in Chapter 5 shows the system formally and in more detail.

In the articles in which Delahayes introduced Ltac [Del00, Del02], the grammar of tactics and expressions is presented formally, but the semantics is only given as an informal big-step description.

As the first approach to describe Ltac in a formal manner, Kirchner proposed a small-step operational semantics for Ltac in the form of a reduction semantics in [Kir03]. He has assumed a rather complex interface of proof context objects and uses fairly complicated side conditions. 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).

In [MGW96] Martin et al. introduce a calculus for the Angel tactical language. While in LCF (and subsequently in Coq) the alternative tactical (denoted \( t_1 \parallel t_2 \) in Coq) commits to the first local success, Angel supports more Prolog-like behavior and incorporates global backtracking. They give a denotational semantics (via the list monad) and prove many equational laws concerning the tactics. In [MG02], a follow-up paper by Martin and Gibbons, the approach to backtracking is generalized by using monads.

In their work on Tinycals [CTZ07], Coen et al. give a small-step operational semantics for the tactic language of Matita [Mat]. They do not relate the semantics to other semantic formats.

In a series of articles about hierarchical proofs known as hiproofs [ADL10, WADG11, DPT06], Aspinall et al. introduce the Hitac tactic language and propose both a big-step and small-step operational semantics for it and prove their equivalence. The authors mention that the design of the small-step semantics was non-trivial and required refining until a satisfactory solution has been found. In the latest work on hiproofs, Whiteside et al. [WADG11] use the semantics to formally justify refactorings of proof scripts.
Chapter 8

An abstract machine for Ltac

8.1 Derivation of the 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 [Lan64, Kri07, Ler90, Pey92], which required skill and experience. In contrast, techniques developed by Danvy et al. [ABDM03, DN01] allow one to obtain abstract machines mechanically, by performing transformations on existing semantics. In this section we take advantage of the technique known as functional correspondence [ABDM03].

The functional correspondence (in the original formulation by Ager et al. [ABDM03]) 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 [Rey72] giving as a result an abstract machine. Recently, Piróg and Biermann have demonstrated in [PB10] 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 7.2 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 continuations):

\[
S_v ::= \text{Let}(G, x, e) : S_v \\
| \text{App}(G, es) : S_v \\
| \text{Args}(G, es) : S_a \\
| \text{Pat}(G, m, e, \vec{cl}) : S_v \\
| \text{EExec}_1(G) : S_v \\
| \text{EExpr}(G) : S_x
\]
The transition relation of the derived abstract machine is shown in Figures 8.1, 8.2, 8.4, and 8.3. Figure 8.1 defines the transitions interpreting expressions and tactics. Figure 8.2 defines the evaluation of function application to multiple arguments. We observe that the machine implements the eval/apply model of function application [MP06] that is inherited from the natural semantics. Figure 8.4 contains the transitions interpreting the stack controlling expression evaluation. Finally, Figure 8.3 displays the transitions interpreting the stack controlling tactic execution.

The following table presents the forms of the configurations of the machine:

<table>
<thead>
<tr>
<th>configuration</th>
<th>judgment</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{E}(G,e,S_v)_v$</td>
<td>$G \triangleright e \downarrow v r_v$</td>
<td>expression evaluation</td>
</tr>
<tr>
<td>$\mathcal{A}(S_v,r_x)_v$</td>
<td>$G \triangleright e \downarrow v r_v$</td>
<td>expression evaluation</td>
</tr>
<tr>
<td>$\mathcal{E}(G,t,S_x)_x$</td>
<td>$G \triangleright t \downarrow x r_x$</td>
<td>tactic execution</td>
</tr>
<tr>
<td>$\mathcal{A}(S_x,r_x)_x$</td>
<td>$G \triangleright t \downarrow x r_x$</td>
<td>tactic execution</td>
</tr>
<tr>
<td>$\mathcal{E}(G,e,S_x)_ee$</td>
<td>$G \triangleright e \downarrow x r_x$</td>
<td>expression execution</td>
</tr>
<tr>
<td>$\mathcal{E}(G,e,S_v)_{ee}$</td>
<td>$G \triangleright e \downarrow x r_v$</td>
<td>extended evaluation</td>
</tr>
<tr>
<td>$\mathcal{E}(G,m,e,\cdot \vec{c},S_v)_{pat}$</td>
<td>$G \triangleright m \cdot e \cdot \vec{c} \downarrow \text{pattern } r_v$</td>
<td>pattern matching</td>
</tr>
<tr>
<td>$\mathcal{E}(G,es,S_a)_{args}$</td>
<td>$G \triangleright es \downarrow \text{args } r_{\text{args}}$</td>
<td>argument evaluation</td>
</tr>
<tr>
<td>$\mathcal{A}(S_a,\cdot \text{args})_{args}$</td>
<td>$G \triangleright es \downarrow \text{args } r_{\text{args}}$</td>
<td>argument evaluation</td>
</tr>
<tr>
<td>$\mathcal{E}(Gs,es,S_x)_{seq}$</td>
<td>$Gs \triangleright es \downarrow \text{seq } r_x$</td>
<td>sequential execution</td>
</tr>
</tbody>
</table>

The initial configurations of the machine are of the form

$$\mathcal{E}(G,e,\text{Nil})_{ee}$$

whereas the final configurations are of the form

$$\mathcal{A}(\text{Nil},r)_x.$$
**Theorem 8.1.** For any goal \( G \) and closed expression \( e \) we have:

\[
G \triangleright e \Downarrow r \iff E(G, e, \text{Nil}) \Rightarrow^* A(\text{Nil}, r),
\]

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 [PB10].

In fact, through the functional correspondence, the abstract machine is not only extensionally but also intentionally 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.

We can also prove that our semantics does not have any missing transitions (so there are no “gaps” in the semantics) provided configurations are closed (i.e. the source code that the machine handles (including the code in the stacks) does not contain any free variables) and valid (i.e. \( |Gs| = |es| \) in configurations and frames concerning sequentialization):

**Theorem 8.2 (Progress).** Any valid closed configuration \( C \) of the machine is either of the form \( A(\text{Nil}, r_x) \) or there exists a valid closed configuration \( C' \) such that \( C \Rightarrow C' \).

**Proof.** By inspection of the transitions.

By Theorem 8.1 we can also conclude that the natural semantics does not have any gaps. It would be however very cumbersome to state Theorem 8.2 directly using the natural semantics, For example, some computations are non-terminating and this requires infinite derivations. Here we can analyze single transitions and the proof itself is easy.

### 8.2 Optimized abstract machine

The abstract machine of this section directly corresponds to the natural semantics of CoreLtac and it has not been 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 [HMP98]. 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 [BBD05], where the presence of a meta-stack (a stack of stacks) supports handling jumps.
\[E(G, v, S_v) v \Rightarrow A(S_v, v)\]
\[E(G, e \ es, S_v) v \Rightarrow E(G, e, App(G, es) : S_v) v\]
\[E(G, let x := e_1 in e_2, S_v) v \Rightarrow E(G, e_1, Let(G, x, e_2) : S_v) v\]
\[E(G, mgoal \varepsilon, S_v) v \Rightarrow A(S_v, \perp_0)\]
\[E(G, mgoal ((p, e) : \vec{cl}), S_v) v \Rightarrow E(G, start G p, e, \vec{cl}, S_v) pat\]
\[E(G, fix x e, S_v) v \Rightarrow E(G, e[x := \text{fix } x e], S_v)\]
\[E(G, idtac, S_x) x \Rightarrow A(S_x, [G]_x)\]
\[E(G, fail n, S_x) x \Rightarrow A(S_x, \perp_n) x\]
\[E(G, progress e, S_x) x \Rightarrow E(G, e, Prog(G) : S_x) ee\]
\[E(G, first \varepsilon, S_x) x \Rightarrow A(S_x, \perp_0)\]
\[E(G, first (e : es), S_x) x \Rightarrow E(G, e, First(G, es) : S_x) ee\]
\[E(G, e_1 ; e_2, S_x) x \Rightarrow E(G, e_1, Semi(e_2) : S_x) ee\]
\[E(G, e_1 ; es, S_x) x \Rightarrow E(G, e_1, BSemi(es) : S_x) ee\]
\[E(G, m, e, \vec{cl}, S_v) pat \Rightarrow E(G, mgoal \vec{cl}, S_v) v\]
\[E(G, m, e, \vec{cl}, S_v) pat \Rightarrow E(G, \sigma(e), Pat(G, e, m', \vec{cl}) : S_v) vx\]
\[E(G, \varepsilon, S_a) args \Rightarrow A(S_a, \varepsilon) args\]
\[E(G, e : es, S_a) args \Rightarrow E(G, e, Args(G, es) : S_a) v\]
\[E(\varepsilon, \varepsilon, S_x) seq \Rightarrow A(S_x, \varepsilon) x\]
\[E(G : Gs, e : es, S_x) seq \Rightarrow E(G, e, Seq(Gs, es) : S_x) ee\]
\[E(G, e, S_v) vx \Rightarrow E(G, e, EEExec(G) : S_v)\]
\[E(G, e, S_x) ee \Rightarrow E(G, e, EExpr(G) : S_x)\]

Figure 8.1: Abstract machine – expression evaluation and tactic execution
\[
\begin{align*}
A(\text{Args}_1(G, \lambda \vec{x}.e) : S_v, \bot_n)_{\text{args}} & \Rightarrow A(S_v, \bot_n)_v \\
A(\text{Args}_1(G, \lambda \vec{x}.e) : S_v, \vec{v})_{\text{args}} & \Rightarrow \\
& \begin{cases}
|\vec{v}| < |\vec{x}| \rightarrow \text{let } (\vec{x}_1, \vec{x}_2) = \text{split } \vec{x} \text{ at } |\vec{v}| \text{ in } \\
A(S_v, \lambda \vec{x}_2.e[\vec{x}_1 := \vec{v}] )_v
\end{cases}
\end{align*}
\]
\[
\begin{align*}
A(\text{Args}_2(v) : S_a, \bot_n)_{\text{args}} & \Rightarrow A(S_a, \bot_n)_{\text{args}} \\
A(\text{Args}_2(v) : S_a, \vec{v})_{\text{args}} & \Rightarrow A(S_a, v : \vec{v})_{\text{args}}
\end{align*}
\]

**Figure 8.2:** Abstract machine – function application

\[
\begin{align*}
A(\text{Prog}(G) : S_x, \bot_n)_x & \Rightarrow A(S_x, \bot_n)_x \\
A(\text{Prog}(G) : S_x, Gs)_x & \Rightarrow A(S_x, \bot_0)_x \quad \text{if } Gs = [G] \\
A(\text{Prog}(G) : S_x, Gs)_x & \Rightarrow A(S_x, Gs)_x \quad \text{if } Gs \neq [G] \\
A(\text{First}(G, es) : S_x, \bot_0)_x & \Rightarrow \text{E}(G, \text{first } es, S_x)_x \\
A(\text{First}(G, es) : S_x, \bot_{\text{args}})_x & \Rightarrow A(S_x, \bot_n)_x \\
A(\text{First}(G, es) : S_x, Gs)_x & \Rightarrow A(S_x, Gs)_x \\
A(\text{Semi}(e) : S_x, \bot_n)_x & \Rightarrow A(S_x, \bot_n)_x \\
A(\text{Semi}(e) : S_x, Gs)_x & \Rightarrow \text{E}(Gs, (\text{args } e), S_x)_\text{seq} \\
A(\text{BSemi}(es) : S_x, \bot_n)_x & \Rightarrow A(S_x, \bot_n)_x \\
A(\text{BSemi}(es) : S_x, Gs)_x & \Rightarrow A(S_x, \bot_0)_x \quad \text{if } |Gs| \neq |es| \\
A(\text{BSemi}(es) : S_x, Gs)_x & \Rightarrow \text{E}(Gs, es, S_x)_\text{seq} \quad \text{if } |Gs| = |es| \\
A(\text{Seq}_1(Gs, es) : S_x, \bot_n)_x & \Rightarrow A(S_x, \bot_n)_x \\
A(\text{Seq}_1(Gs, es) : S_x, Gs')_x & \Rightarrow \text{E}(Gs, es, \text{Seq}_2(Gs') : S_x)_\text{seq} \\
A(\text{Seq}_2(Gs) : S_x, \bot_n)_x & \Rightarrow A(S_x, \bot_n)_x \\
A(\text{Seq}_2(Gs) : S_x, Gs')_x & \Rightarrow A(S_x, Gs' + Gs'')_x \\
A(\text{EEexec}_2 : S_v, r)_x & \Rightarrow A(S_v, r)_v
\end{align*}
\]

**Figure 8.3:** Abstract machine – execution stack
\[
\begin{align*}
\mathcal{A}(\text{Let}(G, x, e_2) : S_v, Gs)_v & \Rightarrow \mathcal{A}(S_v, \bot_0)_v \\
\mathcal{A}(\text{Let}(G, x, e_2) : S_v, \bot_n)_v & \Rightarrow \mathcal{A}(S_v, \bot_n)_v \\
\mathcal{A}(\text{Let}(G, x, e_2) : S_v, v)_v & \Rightarrow \varepsilon(G, e_2[x := v], S_v)_v \\
\mathcal{A}(\text{App}(G, es) : S_v, \bot_n)_v & \Rightarrow \mathcal{A}(S_v, \bot_n)_v \\
\mathcal{A}(\text{App}(G, es) : S_v, Gs)_v & \Rightarrow \mathcal{A}(S_v, v)_v \\
\mathcal{A}(\text{App}(G, es) : S_v, [\bar{n}])_v & \Rightarrow \mathcal{A}(S_v, \bot_0)_v \\
\mathcal{A}(\text{App}(G, es) : S_v, t)_v & \Rightarrow \mathcal{A}(S_v, \bot_0)_v \\
\mathcal{A}(\text{App}(G, es) : S_v, \lambda \bar{x}.e)_v & \Rightarrow \varepsilon(G, es, \text{Args}_1(G, \lambda \bar{x}.e) : S_v)_v \\
\mathcal{A}(\text{Args}(G, es) : S_a, v)_v & \Rightarrow \varepsilon(G, es, \text{Args}_2(v) : S_a)_a \\
\mathcal{A}(\text{Args}(G, es) : S_a, Gs)_v & \Rightarrow \mathcal{A}(S_a, \bot_0)_a \\
\mathcal{A}(\text{Args}(G, es) : S_a, \bot_n)_v & \Rightarrow \mathcal{A}(S_a, \bot_n)_a \\
\mathcal{A}(\text{Pat}(G, m, e, \bar{c}l) : S_v, v)_v & \Rightarrow \mathcal{A}(S_v, v)_v \\
\mathcal{A}(\text{Pat}(G, m, e, \bar{c}l) : S_v, Gs)_v & \Rightarrow \mathcal{A}(S_v, Gs)_v \\
\mathcal{A}(\text{Pat}(G, m, e, \bar{c}l) : S_v, \bot_0)_v & \Rightarrow \varepsilon(G, m, e, \bar{c}l, S_v)_{\text{pat}} \\
\mathcal{A}(\text{Pat}(G, m, e, \bar{c}l) : S_v, \bot_{s(n)})_v & \Rightarrow \mathcal{A}(S_v, \bot_n)_v \\
\mathcal{A}(\text{EExec}_1(G) : S_v, \bot_n)_v & \Rightarrow \mathcal{A}(S_v, \bot_n)_v \\
\mathcal{A}(\text{EExec}_1(G) : S_v, Gs)_v & \Rightarrow \mathcal{A}(S_v, Gs)_v \\
\mathcal{A}(\text{EExec}_1(G) : S_v, [\bar{n}])_v & \Rightarrow \mathcal{A}(S_v, [\bar{n}])_v \\
\mathcal{A}(\text{EExec}_1(G) : S_v, \lambda \bar{x}.e)_v & \Rightarrow \mathcal{A}(S_v, \lambda \bar{x}.e)_v \\
\mathcal{A}(\text{EExec}_1(G) : S_v, t)_v & \Rightarrow \varepsilon(G, t, \text{EExec}_2 : S_v)_x \\
\mathcal{A}(\text{EExpr}(G) : S_x, \bot_n)_v & \Rightarrow \mathcal{A}(S_x, \bot_n)_x \\
\mathcal{A}(\text{EExpr}(G) : S_x, Gs)_v & \Rightarrow \mathcal{A}(S_x, Gs)_x \\
\mathcal{A}(\text{EExpr}(G) : S_x, [\bar{n}])_v & \Rightarrow \mathcal{A}(S_x, \bot_0)_x \\
\mathcal{A}(\text{EExpr}(G) : S_x, \lambda \bar{x}.e)_v & \Rightarrow \mathcal{A}(S_x, \bot_0)_x \\
\mathcal{A}(\text{EExpr}(G) : S_x, t)_v & \Rightarrow \varepsilon(G, t, S_x)_x
\end{align*}
\]

Figure 8.4: Abstract machine – evaluation stack
Chapter 9

A reduction semantics for Ltac

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 Bieracka and Danvy on the syntactic correspondence [BD07] 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 8.1, 8.2, 8.3, and 8.4.

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 Chapter 7. 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:

\[
\begin{align*}
\text{(sequence)} & \quad s ::= \varepsilon \mid Gs \triangleright es \mid (ct : s) \\
\text{(evaluation result)} & \quad r_v ::= r_x \mid v_{eval} \\
\text{(execution result)} & \quad r_x ::= Gs \mid \bot_n
\end{align*}
\]
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 $\texttt{eval ct}$, $\texttt{exec G e}$ and $\texttt{exec 1 G c}$ that serve to make transitions from one computation mode to the other. Specifically, $\texttt{eval ct}$ denotes a closure that is first executed and then the result is used in the evaluation mode, in $\texttt{exec G e}$ the closure is first evaluated and if the result is a tactical, then it is executed. $\texttt{exec 1 G c}$ is used only when evaluating clauses of the $\texttt{mgoal}$ construct.

We have the following reduction contexts:

- **evaluation context** $E_v ::= E_v[\texttt{let } x := [ ] \text{ in } c]$
  - $E_v[[ ] \text{ c}]$
  - $E_v[[ ] : \text{ cs}]$
  - $E_v[\texttt{mgoal } (m, [ ] , \vec{d})]$
  - $E_v[\texttt{exec 1 G } \triangleright [ ]]$
  - $E_v[\texttt{exec G } \triangleright [ ]]$

- **argument context** $E_a ::= E_v[G \triangleright (v_{eval} [ ])]$
  - $E_a[v_{eval} : [ ]]$
(execution context) \[ E_x ::= [] \]
| \[ E_x[\text{progress } G [ ]] \]
| \[ E_x[\text{first } ([ ] : cs)] \]
| \[ E_x[[ ]; e] \]
| \[ E_x[[ ]; es] \]
| \[ E_x[[ ], s] \]
| \[ E_x[Gs > [ ]] \]
| \[ E_v[\text{eval } [ ]] \]

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 9.1 and 9.2.

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 \( E_v[c_1] \rightarrow E_v[c_2] \) if \( c_1 \rightarrow_v c_2 \) and executions of the form \( E_x[ct_1] \rightarrow E_x[ct_2] \) if \( ct_1 \rightarrow_x ct_2 \). We also use auxiliary reductions \( \rightarrow_a \) and \( \rightarrow_s \) to process lists of arguments and sequences of goals and we have \( E_a[cs] \rightarrow E_a[cs'] \) if \( cs \rightarrow_a cs' \) and \( E_x[seq s] \rightarrow E_x[seq s'] \) if \( s \rightarrow_s s' \). The one-step reduction relation \( \rightarrow \) 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 9.1 (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 con-
text (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 9.2.** For any goal $G$ and closed expression $e$ we have:

$$\text{exec } G \triangleright e \rightarrow^* r \quad \text{iff} \quad \mathcal{E}(G, e, \text{Nil}) \triangleright \rangle \Rightarrow^* \mathcal{A} \langle \text{Nil}, r \rangle \langle x, \text{ee} \rangle,$$

where $\rightarrow^*$ is the reflexive-transitive closure of $\rightarrow$. 
\[ (G \triangleright \lambda \vec{x}. e) \vec{v} \rightarrow_v \]

\[
\begin{cases}
|\vec{v}| < |\vec{x}| \rightarrow \text{let } (x_1, x_2) = \text{split } \vec{x} \text{ at } |\vec{v}| \text{ in } \\
\quad \lambda x_2. e[x_1 := \vec{v}] \\
|\vec{v}| = |\vec{x}| \rightarrow G \triangleright e[\vec{x} := \vec{v}] \\
|\vec{v}| > |\vec{x}| \rightarrow \text{let } (v_1, v_2) = \text{split } \vec{v} \text{ at } |\vec{x}| \text{ in } \\
\quad G \triangleright e[\vec{x} := \vec{v}] \quad v_2
\end{cases}
\]

(\text{beta}_v)

\[
G \triangleright \lambda \vec{x}. e \quad \bot_n \rightarrow_v \quad \bot_n
\]

(\text{prop_app})

\[
G \triangleright (e es) \rightarrow_v (G \triangleright e) (G \triangleright es)
\]

(app_l_bot)

\[
\bot_n es \rightarrow_v \bot_n
\]

(prop_l_uval)

\[
c es \rightarrow_v \bot_0 \text{ if } c \neq \bot_n \text{ and } c \neq \lambda \vec{x}. e
\]

(prop_let)

\[
G \triangleright \text{let } x := e_1 \text{ in } e_2 \rightarrow_v \text{let } x := G \triangleright e_1 \text{ in } G \triangleright e_2
\]

(let_gs)

\[
\text{let } x := Gs \text{ in } c \rightarrow_v \bot_0
\]

(let_bot)

\[
\text{let } x := \bot_n \text{ in } c \rightarrow_v \bot_n
\]

(let_v)

\[
\text{let } x := v \text{ in } c \rightarrow_v c[x := v]
\]

(meval)

\[
G \triangleright \text{mgoal } \varepsilon \rightarrow_v \bot_0
\]

(prop_prop)

\[
G \triangleright \text{mgoal } ((p,e) : \vec{c}l) \rightarrow_v \text{mgoal } \text{start } G p, G \triangleright e, \vec{c}l
\]

(prop_l_val)

\[
\text{mgoal } (m,G \triangleright e,\vec{c}l) \rightarrow_v G \triangleright \text{mgoal } \vec{c}l \text{ if next } m = \text{Done}
\]

(prop_match)

\[
\text{mgoal } (m,G \triangleright e,\vec{c}l) \rightarrow_v \text{meval } (m', \text{pat } (G,e) \triangleright (\text{exec}_1 G \triangleright (G \triangleright \sigma(e)))), \vec{c}l)
\]

if next \( m = \text{Match}(\sigma, m') \)

(prop_val)

\[
\text{meval } (m', \text{pat } (G,e) \triangleright v, \vec{c}l) \rightarrow_v v
\]

(prop_gs)

\[
\text{meval } (m', \text{pat } (G,e) \triangleright Gs, \vec{c}l) \rightarrow_v Gs
\]

(prop_bot)

\[
\text{meval } (m', \text{pat } (G,e) \triangleright \bot_0, \vec{c}l) \rightarrow_v \text{mgoal } (m', G \triangleright e, \vec{c}l)
\]

(prop_botS)

\[
\text{meval } (m', \text{pat } (G,e) \triangleright \bot_S n, \vec{c}l) \rightarrow_v \bot_n
\]

(prop_fix)

\[
G \triangleright \text{fix } x \ e \rightarrow_v G \triangleright e[x := \text{fix } x \ e]
\]

(prop_exec1_tac)

\[
\text{exec}_1 G \triangleright t \rightarrow_v \text{eval } (G \triangleright t)
\]

(prop_exec1_res)

\[
\text{exec}_1 G \triangleright r_x \rightarrow_v r_x
\]

(prop_exec1_val)

\[
\text{exec}_1 G \triangleright v \rightarrow_v v \text{ if } v \neq t
\]

(prop_eval_res)

\[
\text{eval } r_x \rightarrow_v r_x
\]

(prop_goal_val)

\[
G \triangleright v \rightarrow_v v
\]
Figure 9.2: Reduction semantics – execution and auxiliary contractions
Part IV

Typed Tactical Languages
Chapter 10

A type system for Ltac

In the semantics for Ltac presented in previous chapter we saw that Ltac can be seen as a dynamically typed programming language and that type errors are handled in the same way as tactic invocation failures. We believe that, in practice, the former does not add much to the expressive power of the language. Rather it allows many opportunities for bugs to occur.

The semantics we have seen so far are designed so that the progress property (Theorem 8.1) is maintained. For this end, some rules have to deal with situations that would be deemed as type errors in a typed setting. In this chapter we want to identify and remove all such rules. To do so – without losing the progress property! – we need to ban invalid (or “stuck”) configurations. We will achieve this by designing a type system. The system will include a “user view”, that is a set of rules that concerns only expressions and tactics (ie. the syntax used by the users), which will be next extended to handle intermediate states of computations – in our case stacks and configurations, as we found the abstract machine to be the most convenient; most often a reduction semantics is used for this end.

10.1 Failures that originate from type errors

The following rules of the natural semantics deal with typing errors:

\[ \frac{G \triangleright e \downarrow_v v \quad v = \left[ n \right] \mid \lambda \vec{x}. e}{G \triangleright e \downarrow \perp_0} \frac{G \triangleright e \downarrow_v v \quad v = \left[ n \right] \mid t}{G \triangleright e \ es \downarrow_v \perp_0} \]

\[ \frac{G \triangleright e \downarrow_v Gs}{G \triangleright (e : es) \downarrow_{args} \perp_0} \frac{G \triangleright e_1 \downarrow_v Gs}{G \triangleright \text{let } x := e_1 \text{ in } e_2 \downarrow_v \perp_0} \]

The \( \text{EEX}_4 \) rule is used to ensure that tacticals only invoke tactics. The other rules are concerned with function application and local binding: \( \text{APP}_2 \) signals an error when a non-lambda value appears in the function position, while
rules \texttt{ARGS}, and \texttt{LET}, ensure that only values (expressions in case of \texttt{fix}) can be passed as arguments or bound to local names.

### 10.2 Type system for expressions

We have the following types for expressions:

\[
\begin{align*}
\text{(value type)} & \quad \nu ::= \mathbb{N} | \text{Tac} | \nu \rightarrow \tau \\
\text{(general type)} & \quad \tau ::= \nu | \text{Goal}
\end{align*}
\]

We also use the following function on general types:

\[
\begin{align*}
\text{force Tac} &= \text{Goal} \\
\text{force } \tau &= \tau \text{ if } \tau \neq \text{Tac}
\end{align*}
\]

The type system for expressions is presented in Figure 10.1 and the judgment defined here has the form 

\[
e : \tau
\]

In contrast to previous presentations of type systems in the thesis, we use a style in which only the locally used assumptions are mentioned using the \( x : \nu \vdash e : \tau \) notation. The difference is purely notational and it is easy to recover the previous style.

As we have seen in the natural semantics, only values can be passed as function arguments or bound to a local variable. Therefore we restrict the function types so that non-values can appear only in strictly positive positions. The feature that complicates the type system the most is – yet again – the \texttt{match goal} construction. The problem is that \texttt{match goal} executes tactics even in the evaluation mode. As a consequence we can never obtain a tactic from the evaluation of a \texttt{match goal} – this fact is reflected in the type system by the use of the \texttt{MGOAL} rule. This allows us to statically prevent the awkward interaction/pitfall between \texttt{match goal} and \texttt{let} (which also occurs when \texttt{match goal} is given as a function argument). Yet, the \texttt{force} function has effect if and only if the input type equals \texttt{Tac}, so while the script

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

will be consider ill-typed, examples in which \texttt{match goal} returns an integer or a function will continue to work, as long as the type of each branch is the same.

We should note that expressions that are “pure” tactics have type \texttt{Tac}, while tactics wrapped with \texttt{match goal} will have type \texttt{Goal}. Both kinds of
expressions act like tactics when executed, so we have introduced an auxiliary judgment \( e \text{ tactic} \), which is provable when the type of \( e \) is either \( \text{Tac} \) or \( \text{Goal} \). Intuitively, \( e \text{ tactic} \) means that evaluation of \( e \) will yield either a tactic or a list of subgoals.

### 10.3 A type system for abstract machine configurations

While the user is only interested in typing of expressions, we have to account for the configurations of the abstract machine. We must therefore provide typing rules for both the stacks and the configurations of the machine. The rules for the stacks are presented in Figure 10.2, while the rules for configurations are presented in Figure 10.4. We also need to assign types to execution results – the rules are shown in Figure 10.3.

We now proceed to discuss the design of the type system for the intermediate configurations. First of all, we need to be able to connect the type of the expression the machine is currently reducing with the current stack. Moreover, not all syntactically correct stacks are valid, as the values kept in frames should satisfy certain invariants. Therefore, Figure 10.2 introduces the judgment

\[
S \trianglelefteq \tau
\]

The intended meaning is that the stack \( S \) is valid and that the current expression should yield a result of type \( \tau \). However, because the \( \mathcal{E}(G,e,S_a)_{\text{args}} \) helper configuration needs to handle lists of values, we have to introduce internal types denoting type list constructors

\[
\tau ::= \ldots \mid \varepsilon \quad \text{empty list} \mid (\nu : \tau) \quad \text{consing}
\]

When convenient we use \( [\nu_1, \ldots, \nu_n] \) to denote an \( n \)-element list of types. We also generalize the typing judgment to allow us to say \( \bar{v} : \bar{\nu} \) instead of the cumbersome \( |\bar{v}| = |\bar{\nu}| \land \forall i \in [1,|\bar{v}|], \bar{v}[i] = \bar{\nu}[i] \). The generalized judgment is inductively defined as

\[
\begin{align*}
\text{NIL} & \quad \varepsilon : \varepsilon \\
\text{CONS} & \quad v : \nu \quad \bar{v} : \bar{\nu} \\
& \quad \frac{}{v : \nu \quad \bar{v} : \bar{\nu} \quad \left( v : \nu \right) : \left( \bar{v} : \bar{\nu} \right)}
\end{align*}
\]

At this point we can assign types to both expressions and stacks, so Figure 10.4 defines the judgment

\[
C \ \text{ok}
\]

The intended meaning is that the configuration \( C \) is valid and represents an intermediate state of computation of a type correct expression.

The design of the system is driven by the typing rules for expressions and our desire to have the preservation property, introduced in the next section.
<table>
<thead>
<tr>
<th>Rule</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{VAR}</td>
<td>\nu \vdash x : \nu</td>
</tr>
<tr>
<td>\text{LIT}</td>
<td>\nu \vdash \lceil n \rceil : \mathbb{N}</td>
</tr>
<tr>
<td>\text{ABS}</td>
<td>\nu_1, \ldots, \nu_n \vdash e : \tau</td>
</tr>
<tr>
<td>\text{APP}</td>
<td>\nu \vdash e_1, \ldots, e_n : \tau</td>
</tr>
<tr>
<td>\text{LET}</td>
<td>\nu \vdash x : \nu \vdash e_2 : \tau</td>
</tr>
<tr>
<td>\text{MGOAL}</td>
<td>\forall (p, e) \in \overrightarrow{\text{cl}}. e : \nu</td>
</tr>
<tr>
<td>\text{FIX}</td>
<td>\nu \vdash e : \tau</td>
</tr>
<tr>
<td>\text{IDTAC}</td>
<td>\text{idtac : Tac}</td>
</tr>
<tr>
<td>\text{FAIL}</td>
<td>a : \mathbb{N}</td>
</tr>
<tr>
<td>\text{SEMI}</td>
<td>e_1 \text{ tactic}, e_2 \text{ tactic}</td>
</tr>
<tr>
<td>\text{BRANCH}</td>
<td>\nu \vdash e : \tau</td>
</tr>
<tr>
<td>\text{FIRST}</td>
<td>\nu \vdash e : \tau</td>
</tr>
<tr>
<td>\text{PROGRESS}</td>
<td>e \text{ tactic}</td>
</tr>
<tr>
<td>\text{TACTIC}</td>
<td>e : \tau</td>
</tr>
</tbody>
</table>

\textbf{Figure 10.1:} Type system – expressions and tactics
NIL: \[ \text{Nil} \vdash \text{Goal} \]

PROG: \[ S_x \vdash \text{Goal} \]
\[ \text{Prog}(G) : S_x \vdash \text{Goal} \]

FIRST: \[ S_x \vdash \text{Goal} \]
\[ \forall e \in es, e \cdot \text{tactic} \]
\[ \text{First}(G, es) : S_x \vdash \text{Goal} \]

SEMI: \[ S_x \vdash \text{Goal} \]
\[ e \cdot \text{tactic} \]
\[ \text{Semi}(e) : S_x \vdash \text{Goal} \]

BSEMI: \[ S_x \vdash \text{Goal} \]
\[ \forall e \in es, e \cdot \text{tactic} \]
\[ \text{BSemi}(es) : S_x \vdash \text{Goal} \]

SEQ1: \[ S_x \vdash \text{Goal} \]
\[ \forall e \in es, e \cdot \text{tactic} \]
\[ |Gs| = |es| \]
\[ \text{Seq}_1(Gs, es) : S_x \vdash \text{Goal} \]

SEQ2: \[ S_x \vdash \text{Goal} \]
\[ \text{Seq}_2(Gs) : S_x \vdash \text{Goal} \]

EEXEC2: \[ S_v \vdash \tau \]
\[ x : \nu \vdash e : \tau \]
\[ \text{Let}(G, x, e) : S_v \vdash \nu \]

APP: \[ S_v \vdash \tau \]
\[ es : [\nu_1, \ldots, \nu_n] \]
\[ \text{App}(G, es) : S_v \vdash \nu_1 \rightarrow \cdots \rightarrow \nu_n \rightarrow \tau \]

ARGS: \[ S_a \vdash (\nu : \vec{\nu}) \]
\[ es : \vec{\nu} \]
\[ \text{Args}(G, es) : S_a \vdash \vec{\nu} \]

PAT: \[ S_v \vdash \text{force} \tau \]
\[ e : \tau \]
\[ \forall (p, e) \in \vec{cl}, e : \tau \]
\[ \text{Pat}(G, m, e, \vec{cl}) : S_v \vdash \text{force} \tau \]

EEXEC: \[ S_v \vdash \text{force} \tau \]
\[ \text{EExec}_1(G) : S_v \vdash \tau \]

EE1: \[ S_x \vdash \text{Goal} \]
\[ \text{EExpr}(G) : S_x \vdash \text{Goal} \]

EE2: \[ S_x \vdash \text{Goal} \]
\[ \text{EExpr}(G) : S_x \vdash \text{Tac} \]

ARGS1: \[ S_v \vdash \tau \]
\[ \lambda \vec{x}. e : \nu_1 \rightarrow \cdots \rightarrow \nu_n \rightarrow \tau \]
\[ \text{Args}_1(G, \lambda \vec{x}. e) : S_v \vdash [\nu_1, \ldots, \nu_n] \]

ARGS2: \[ S_a \vdash (\nu : \vec{\nu}) \]
\[ v : \nu \]
\[ \text{Args}_2(v) : S_a \vdash \vec{\nu} \]

Figure 10.2: Type system – machine stacks
Figure 10.3: Type system – results

\[
\begin{align*}
\text{GOALS} & : G s : \text{Goal} \\
\text{BOT} & : \bot_n : \tau
\end{align*}
\]

Figure 10.4: Type system – configurations
10.4 Type safety

In this section we want to simplify the semantics for CoreLtac. We therefore introduce a variant of the original abstract machine.

Definition 10.1. The complete abstract machine is the abstract machine for CoreLtac presented in chapter 8.

Definition 10.2. The simplified abstract machine is the complete abstract machine with the following (type-error handling) transitions removed:

\[
\begin{align*}
\mathcal{A}\langle \text{Let}(G, x, e_2) : S_v, Gs\rangle_v & \Rightarrow \mathcal{A}\langle S_v, \bot_0\rangle_v \\
\mathcal{A}\langle \text{App}(G, es) : S_v, Gs\rangle_v & \Rightarrow \mathcal{A}\langle S_v, \bot_0\rangle_v \\
\mathcal{A}\langle \text{App}(G, es) : S_v, n\rangle_v & \Rightarrow \mathcal{A}\langle S_v, \bot_0\rangle_v \\
\mathcal{A}\langle \text{App}(G, es) : S_v, t\rangle_v & \Rightarrow \mathcal{A}\langle S_v, \bot_0\rangle_v \\
\mathcal{A}\langle \text{Args}(G, es) : S_a, Gs\rangle_v & \Rightarrow \mathcal{A}\langle S_a, \bot_0\rangle_{args} \\
\mathcal{A}\langle \text{EExpr}(G) : S_x, n\rangle_v & \Rightarrow \mathcal{A}\langle S_x, \bot_0\rangle_x \\
\mathcal{A}\langle \text{EExpr}(G) : S_x, \lambda\vec{x}.e\rangle_v & \Rightarrow \mathcal{A}\langle S_x, \bot_0\rangle_x
\end{align*}
\]

Now we want to prove that our type system is designed correctly with respect to the simplified abstract machine. For this end we prove the progress and preservation properties.

Theorem 10.3 (Progress). For any configuration \(C, C_\text{ok}\) implies that either \(C\) is of the form \(\mathcal{A}\langle \text{Nil}, r_x\rangle_x\) or there exists a configuration \(C'\) such that \(C \Rightarrow C'\).

Proof. We have previously verified the progress property for the complete abstract machine (Theorem 8.2). It remains therefore to see that for every transition missing from the simplified abstract machine (Definition 10.2) the left-hand-side configuration is ill-typed. \(\square\)

Theorem 10.4 (Preservation). \(C_\text{ok}\) and \(C \Rightarrow C'\) imply \(C'_\text{ok}\).

Proof. Proof by inspection of the transition rules and inversion on the typing rules. The cases for application and let require an appeal to the substitution lemma, while case for pattern matching requires the use of the pattern-substitution assumption. \(\square\)

Finally note that we no longer need to mention that the configurations are closed because this is implied by well-typedness.

Corollary 10.5 (Type safety\(^1\)). \(e\) tactic implies that if we start the abstract machine in the \(\mathcal{E}\langle G, e, \text{Nil}\rangle_{ee}\) configuration then it will either stop in the \(\mathcal{A}\langle \text{Nil}, r_x\rangle_x\) configuration or the execution will never finish. In particular, it is not possible for the machine become stuck in a non-final configuration.

\(^1\)adapted to tactic execution
10.5 Summary

We have presented a simple type system for Ltac, possibly realistic enough to be implemented in Coq without any major changes to the existing codebase. The system is somewhat too prohibitive, but it should be a good start and could become a base for a more sophisticated system. For example, it should be noted that some rules seem a bit too complicated. We believe that is not only because the design of the type system has room for improvement, but also the underlying tactical language could undergo simplifications in the semantics. A clean tactical language has a better chance to be accompanied with a clean (and clear) typing system.

The shortcomings of untyped tactic languages have been the motivation on the design of many novel approaches to tactic programming. Compared to our type system, Mtac [ZDK+13] and VeriML [SS10] also allow typed tactic programming, but they serve a different purpose: their goal is to certify reflection-based decision procedures, while we intend to use Ltac as a “glue” that makes it possible to combine various approaches. Therefore our type system is much simpler and our goal much more humble: we only want to make sure that Ltac scripts do not contain embarrassing type errors and the scripts do not fail in strange ways.

In his PhD thesis [Kir07], Kirchner proposes a type system for tactics. Compared to our work, he considers only simple tacticals and his approach is much more restrictive, because his idea is to classify some tactic failures as type errors.
Part V

Conclusion and references
Chapter 11

Conclusion

11.1 Summary

In this thesis we have given an operational account for various aspects of interactive theorem proving: from the proof engine through atomic tactics to the tactic language. For the latter we have developed a versatile semantic toolbox, in the form of a set of three semantic formats: a natural semantics, an abstract machine and a reduction semantics. The semantics are intentionally equivalent, yet their formats differ, so they prove useful in different context and applications. Concretely, we have used the abstract machine to prove a progress result for untyped Ltac and to design and verify a typing system for Ltac. It should be noted however, that in this thesis we have only begun to take advantage of the operational foundation.

11.2 Perspectives

The goal of this thesis has been to develop a semantical framework for the tactic language of Coq. Now that we have the semantics, what could we use them for?

First of all, natural semantics is often used as the root of the notion of program equivalence [Pit97]. Our semantics could be the base of an analogous notion of tactic equivalence, which could open the possibilities of tactic simplifications, refactorizations and optimizations. On the other hand, reduction semantics could be the base of debugger for tactic execution tracing. Such a tool would be useful in those cases, when tactics seem to mysteriously fail. The abstract machine (possibly after optimization) could in turn be used as an reference in the development of a compiler for tactics or for an efficient implementation in a low-level programming language.

Many of the failures arising during tactic execution turn out to be simple type errors (often after tedious and exhaustive bug fixing.) While the type system from Chapter 10 seems to be pretty usable, this claim has yet to
be verified in practice. So, it would be interesting to implement (in Coq) a **type checker** based on the type system we have developed.

While *CoreLtac* is pretty expressive, there are still many features present in *Ltac* (as implemented in contemporary Coq) that have not been addressed here. Most importantly, we have not described the **interaction between *Ltac* and *Gallina*** in full detail and we did not formalize the dependent type theory used as Coq’s logic. This includes the notion of **existential variables** and their use for **proof automation**.

Finally, it should be noted that through this work we have came across many subtleties, gimmicks and idiosyncrasies of *Ltac*, many of which can be classified as pitfalls or even language warts. Those deficiencies manifest themselves in the semantics in the form of unnatural or (unnecessarily) complicated rules. So a different direction for further research is to **design a tactic language that will have a cleaner semantics** and in which certain interactions will be clarified. Based on preliminary results, we believe that a **type system based on modal logic** and its connection to staged computation [DP01] may be called for in this case.
Bibliography


Olivier Danvy and Lasse R. Nielsen. Syntactic theories in practice. In Mark van den Brand and Rakesh M. Verma, editors,


BIBLIOGRAPHY


Appendix A

Streszczenie

W niniejszej pracy podejmuję temat formalizacji interaktywnego dowodzenia twierdzeń za pomocą asystentów dowodzenia. Asystent dowodzenia to program komputerowy, który pozwala na modelowanie pewnych zagadnień w sposób ścisły (za pomocą logiki formalnej implementowanej przez narzędzie) oraz na wyrażanie i dowodzenie własności owego modelu. W pracy zajmuję się narzędziem o nazwie Coq, ale przede wszystkim interesuje mnie Ltac, jeden z języków w dostępnych w Coqu. Ltac służy do programowania taktyk, czyli komend które pozwalają na dowodzenie w sposób formalny a jednocześnie przyjazny dla użytkownika: dzięki taktykom komputerowe dowodzenie twierdzeń staje się przystępne także dla osób, które nie są biegłe z logiki formalnej.

W ciągu ostatniej dekady zostało wykazane, że biężąca technologia umożliwia formalizowanie w asystentach dowodzenia matematycznych dowodów imponujących rozmiarów i o dużym znaczeniu (formalizacja Gonthiera dowodu twierdzenia o kolorowaniu mapy) a także że możliwa jest certyfikacja oprogramowania o dużym znaczeniu (komilator języka C napisany przez Leroy). Raporty techniczne napisane przez autorów projektów o podobnej skali świadczą, że pracy nad tak dużymi formalizacjami towarzyszy duży narzut czasowy przy wykonywaniu czynności takich jak modyfikowanie struktury skryptów z dowodami. Oznacza to, że jest duże zapotrzebowanie na narzędzia, które np. umożliwią sprawne wykonywanie zmian, które nie mają wpływu na poprawność samego dowodu, ale ułatwiają pracę innym bądź po prostu wprowadzają porządek w plikach. Aby zapewnić poprawność takich narzędzi, należy najpierw dysponować m.in. formalnym opisem języka taktyk.

Praca powstała ponieważ zidentyfikowany został brak literatury, która traktowałaoby Ltaca (ogólne: interaktywne dowodzenie twierdzeń w Coqu) w sposób ścisły poprzez zadanie formalnej semantyki (czyli matematycznego opisu). Istniejąca dokumentacja systemu Coq opisuje Ltaca raczej pobieżnie i wybiórczo, zaś jedyna jak dotąd próba zbadania Ltaca (trak-
towanego jako język programowania) została podjęta ponad 10 lat temu przez Flaurenta Kirchnera. Co więcej, głównym tematem pracy Kirchnera nie był sam Ltac a zagadnienie przenaszaalności dowodów między narzędziami. Co gorsze, od tamtej pory Ltac znacząco ewoluował i opis Kirchnera stracił na aktualności.

Zadaniem w niniejszej pracy było zadanie semantyki operacyjnej dla Ltaca. Aby przygotować czytelnika niezaznajomionego z zagadnienie interaktywnego dowodzenia twierdzeń, początkowe rozdziały pracy formalizują kontekst, w którym należy rozpatrywać Ltac, a mianowicie:

- Rozdział 2. opisuje główną strukturę silnika dowodowego (ang. proof engine),
- Rozdział 3. przypomina system dedukcji naturalnej i proponuje przykładowy zestaw taktyk atomowych dla tego systemu,
- Rozdział 4. omawia tematykę wiarygodności dowodów weryfikowanych przez komputer oraz analizuje podejścia do reprezentacji dowodów

Po wyłożeniu fundamentów przechodzimy do głównego tematu, to jest semantyki taktyk. W rozdziałach 5-7 przechodzimy od prostego języka taktyk wyższego rzędu do (podzbioru) Ltaca, w każdym z rozdziałów zadając semantykę naturalną rozpatrywanego języka. Następnie wykorzystujemy dobrze znane techniki transformacji semantyk do otrzymania nowych (równoważnych) semantyk dla Ltaca:

- W rozdziale 8. wykorzystujemy odpowiedniość funkcyjną (ang. functional correspondence) by otrzymać maszynę abstrakcyjną
- W rozdziale 9. wykorzystujemy odpowiedniość syntaktyczną (ang. syntactic correspondence) by otrzymać semantykę redukcyjną

Semantyki z rodzajów 7-9 tworzą formalny fundament, na którym można oprzeć wiele dalszych badań. W rozdziale 10. prezentuję system typów dla Ltaca, który pozwala wyeliminować część niedogodności Ltaca, przede wszystkim te związane z faktem, że Ltac jest dynamicznie typowany. Fundament semantyk operacyjne wypracowany w niniejszym dokumencie może stanowić punkt wyjścia do wielu kierunków badań, część z nich omawiamy w rozdziale 11.

Do pracy dołączona jest implementacja prototypowego silnika dowodowego o nazwie RoCoQo.