MLtac: raising Ltac to the power of ML

Arthur Charguéraud

Coq meeting on tactics June 30th, 2009

Overview of the talk

1– Ltac: its strengths and weaknesses

2– MLtac: a proposal for a new language for tactics

3– Perspectives offered by MLtac

Part 1
Working with Ltac

Experience with Ltac

Three main projects:

– Formal programming language metatheory

– Deep embedding of purely functional code

– An extended set of general-purpose tactics

None of these three projects could have been successful without the ability to program tactics.

I am glad I could avoid programming directly in OCaml, as the cost of entry is high, and the distribution of scripts involving OCaml-based tactics poses problems.

Experience with Ltac

Three main use of Ltac:

– Factorize common sequences of tactics

– Help instantiate non syntax-directed lemmas

– Implement mini-decision procedures

I've learned to implement things that I initially though were impossible to implement in Ltac... yet, I'm getting tired of hacking so much to implement my tactics.

Strengths of Ltac

- Lightweight integration within Coq scripts
- Parsing facilities for Coq terms
- Support for a tactic notation system
- Backtracking pattern matching

Weaknesses of Ltac

- No data types (must hack everything into Coq types)
- No type-checking, virtually impossible to debug (painful)
- Poor support for exceptions (only exception levels)
- No first-class representation of goals nor contexts (limit.)
- No pattern matching under binders (severe limitation)
- No simple non-backtracking matching (must encode it)
- Ambiguity between tactics and lazy tactics (unintuitive)
- Relatively inefficient implementation, no side-effects
- Yet another language to learn, high cost of entry

Part 2
A proposal: MLtac

My dream: MLtac

- Rely on a true programming language, say OCaml
  - → already known by most users
  - → features data structures, exceptions, side-effects
  - → comes with a type checker
  - → allows for efficient execution (bytecode and native)
- In the same time, keep all the benefits of Ltac
  - → lightweight integration in vernacular files
  - → parsing facilities accross terms, tactics, and Caml
  - → pattern matching facilities
  - → efficient support for backtracking

Comparison with current ML tactics

It is already possible to program tactics in OCaml, but:

- one works directly with the representation of terms used in the implementation of Coq :
  - ugly details of the implementation may appear,
  - backward compatibility is very hard to ensure,
  - support for cross-parsing is limited.
- there is no documentation to programming ML tactics,
- it is hard to debug Caml tactics (uses many Obj.magic)
- it is hard to interleave Coq and tactic definitions,
- there is no easy way of distributing scripts :
  - users must be able to compile Coq from sources,
  - no protection mechanism w.r.t. third-party code.

Implementation of MLtac

1) "External" approach: embark a version of OCaml's bytecode compiler and virtual machine within Coqtop. In this setting, the runtime of Coq is totally separated from the runtime of MLtac.

2) "Internal" approach: compile tactics using the standard Ocaml's compiler, and dynamically load the compiled code from Coqtop.

Comparison: it is probably

– easier to enforce security properties with (1)

– easier to support efficient native code tactics with (2)

– hard to load bytecode from a natively-compiled Coq in(2)

Connection with external processes

MLtac would presumably be able to support the many programs, yet it is sometimes useful to be able to rely on some existing external program.

Under the appropriate security policy, it is possible to call an external program from MLtac.

Data can be exchanged in several ways:

– direct link of programs (for internal implementation)

– exchange via files / streams / network sockets / ...

Syntax and semantics of MLtac

Programming in MLtac is programming in ML plus:

– term:(...), ml:(...), tac:(...), pat:(...)
  can be used for selecting the scope
– the scope of identifiers is computed automatically
– special support for pattern matching constructions
– access to primitive functions, e.g. "evar" and "unify"
– side-effects are allowed, but only locally
– exceptions and exception-handlers are supported

Semantics of the ML part is standard, except the linear pattern matching, and the backtracking pattern matching.

An abstract layer for terms

MLtac does not manipulate raw Coq terms from the implementation, but an abstract presentation of them.

A key improvement over Ltac: the treatment of binders. We can adopt an approach where the users explicitly opens and closes abstractions and products. Example:

```
let swap_head T :=
  match T with (forall (x:A) (y:B), T1) =>
    constr:(forall (y:B) (x:A), T1)
```

Representation of goals

- **`type goal`** : an abstract type that describes a goal.

To be detailed: the treatment of unification variables.

Example of functions:

- **`unify : goal -> tuning -> term -> term -> goal`**
- **`evar : goal -> term -> goal * term`**

A tactic is a function of type **`goal -> goal`** (+ exceptions)

The top-level goal is represented as a mutable goal,

which is updated by the function "invoke"

- **`invoke : (goal -> goal) -> unit`**

Part 3
MLtac: perspectives

Beyond tactics programming

Rather than implementing hundreds of feature wishes, implement just a single one:

user-defined top-level commands

This is not hard to do, once Coq already embarks a general-purpose programming language.

It would allow the user to:

– maintain several "states" for use by his tactics,

– implement tools generating definitions and lemmas,

– program ad-hoc parsers and printers for his datatypes,

– ...

This would offer so many possibilities...

A few examples of top-level tools

– generation of definitions and lemmas (e.g. beq, subterm)

– manipulation of hint databases (e.g. merging bases)

– advanced control of notation (specialized parsing/printing for particular datatypes, such as nat, int, real, string, and especially for embedded syntaxes)

– control of goal display (temporary hiding of large local definitions in contexts, well-organized contexts with variables sorted by type)

– fine-grained control of semi-automatic name generation

– labelling of cases and subcases in proofs

...