Tactics and certificates in Meta-Dedukti

Raphaël Cauderlier

March 2, 2017

Outline

Why Tactics?

- short
- several goals at once
- easy to develop interactively
- domain-specific automation
- axiom elimination
- transfer
- checking certificates

Tactics are short

```
Ltac mytactic := simpl; f_equal; assumption.

Lemma plus_commute m n : m + n = n + m.
Proof.
  induction m.
  - induction n.
    + reflexivity.
    + mytactic.
  - transitivity (S (n + m)).
    + mytactic.
    + clear IHm.
      induction n.
      * reflexivity.
      * mytactic.
Defined.
```

Tactics are short

```
fun m n : nat =>
nat_ind (fun m0 : nat => m0 + n = n + m0)
  (nat_ind (fun n0 : nat => 0 + n0 = n0 + 0) eq_refl
     (fun (n0 : nat) (IHn : 0 + n0 = n0 + 0) =>
      (fun H : n0 = n0 + 0 =>
       (fun H0 : n0 = n0 + 0 =>
        eq_trans (f_equal (fun f : nat -> nat => f n0) e
          (f_equal S H0)) H) IHn) n)
  (fun (m0 : nat) (IHm : m0 + n = n + m0) =>
   eq_trans
     ((fun H : m0 + n = n + m0 =>
       (fun H0 : m0 + n = n + m0 =>
        eq_trans (f_equal (fun f : nat -> nat => f (m0 +
          (f_equal S H0)) H) IHm)
     (nat_ind (fun n0 : nat => S (n0 + m0) = n0 + S m0)
        (fun (n0 : nat) (IHn : S (n0 + m0) = n0 + S m0)
         (fun H : S (n0 + m0) = n0 + S m0 =>
          (fun H0 : S (n0 + m0) = n0 + S m0 =>
```

Tactics are easy to develop interactively

```
Ltac mytactic :=
  simpl;
  f_equal;
  assumption.

Lemma plus_commute m n :
   m + n = n + m.
Proof.
  induction m.
```

```
2 subgoals, subgoal 1

  n : nat
  ================
   0 + n = n + 0

subgoal 2 (ID 14) is:
 S m + n = n + S m
```

Tactics allow domain-specific automation

- encode separation logic
- write a tactic proving goals of the form
  $(A * x \mapsto a * B * y \mapsto b * C) \rightarrow x \neq y$

Tactics can serve axiom elimination

Eliminating axiom ax is the same as proving formulae of the form $(\mathsf{ax} \to A) \to A$.

Tactics can be used for transferring theorems

Reasoning modulo isomorphism

Let A and B be two isomorphic structures. If φ_A is a theorem about A then φ_B holds.

We want a `transfer` tactic proving goals of the form $\varphi_A \to \varphi_B$.

Tactics can be used for transferring theorems

Reasoning modulo isomorphism

Let A and B be two isomorphic structures. If φ_A is a theorem about A then φ_B holds.

We want a `transfer` tactic proving goals of the form $\varphi_A \to \varphi_B$.

- On April 27th, Théo Zimmermann is going to talk about this in Coq
- On May 4th, I am going to talk about adapting his work in Meta Dedukti

Tactic are required for certificate checking

Dedukti proofs for Zenon Arith, VeriT, Zipperposition?

- Zenon Arith: reasoning modulo associativity and commutativity of addition (`ring`)
- VeriT: reasoning modulo symmetry of equality (`congruence`)
- Zipperposition: `resolution(A, B, C)`

$$\frac{A \qquad B}{C}(\text{resolution})$$

if there are C_1, C_2, l, l', σ such that

- $A =_{AC} C_1 \vee l$,
- $B =_{AC} C_2 \vee \neg l'$,
- $C =_{AC} \sigma(C_1 \vee C_2)$,
- $\sigma l = \sigma l'$.

Tactic languages

	Tactic language	Type System	Term embedding
LCF	Implementation language	ML	Deep
Ltac	Extra language	Untyped	Shallow
Mtac	Coq Refiner language	Dependent	Mostly Shallow
Lean	Lean	Dependent	Deep
Oyster2	Oyster2	Dependent	Shallow
ACL2	ACL2	Untyped	Deep

All of them:

- handle backtracking
- allow non termination
- do not compromise logical consistency

Non-termination without compromising consistency

Lean approach:

- any symbol flagged as meta is not passed to the kernel
- any symbol depending on a meta symbol should itself be meta
- termination check is deactivated in the meta world

Non-termination without compromising consistency

Mtac approach:

- tactics are kept in a monad
- tactic reduction is not used for conversion
- `run (m : M A)` has type `A` iff `m` tactic-reduces to a term `ret a`
  - in this case, `run m` is replaced by `a`
  - `run` is never seen by the kernel

Non-termination without compromising consistency

Meta Dedukti approach:

- define a good and several bad (not good) rewrite systems
- normalize using bad systems until type checking succeeds using the good system

good $\subset$ {terminating, confluent, consistent, constructive}

Dktactics

Tactic and certificate languages for first-order logic in Meta Dedukti.

- Written (almost) only in Dedukti
- No modification of Dedukti itself
  - No implicit parameter, quoting, or unification
  - Intensive use of Miller pattern and non-linearity
- Easy to adapt to other logics

Encoding First-Order Logic in Dedukti

```
type : Type.
term : type -> Type.
function : Type.
def fun_arity : function -> list type.
def fun_codomain : function -> type.
apply_fun : f : function ->
            hlist (fun_arity f) term ->
            term (fun_codomain f).
```

Encoding First-Order Logic in Dedukti

```
prop : Type.
false : prop.
and : prop -> prop -> prop.
...
ex : A : type -> (term A -> prop) -> prop.

predicate : Type.
def pred_arity : predicate -> list type.
apply_pred : p : predicate ->
             hlist (pred_arity p) term ->
             prop.
```

Encoding First-Order Logic in Dedukti

```
def proof : prop -> Type.
[]    proof false --> c : prop -> proof c
[a,b] proof (and a b) -->
        c : prop ->
        (proof a -> proof b -> proof c) ->
        proof c
[a,b] proof (or a b) -->
        c : prop ->
        (proof a -> proof c) ->
        (proof b -> proof c) ->
        proof c
[a,b] proof (imp a b) --> proof a -> proof b
[A,p] proof (all A p) --> a : term A -> proof (p a)
[A,p] proof (ex A p) -->
        c : prop ->
        (a : term A -> proof (p a) -> proof c) ->
        proof c.
```

The tactic language

- dependently typed
- backtracking
- not linear, not confluent, not consistent

The tactic language

`tactic A` is the type of tactics proving `A` or failing to do so.
`tactic` is a monad.

The tactic language

```
exception : Type.
tactic : prop -> Type.

ret : A : prop -> proof A -> tactic A.
raise : A : prop -> exception -> tactic A.

def run : A : prop -> tactic A -> proof A.
[A,a] run A (ret _ a) --> a.
```

The tactic language

```
def bind : A : prop -> B : prop ->
  tactic A -> (proof A -> tactic B) -> tactic B.
[a,f,t] bind _ _ (ret _ t) f --> f t
[B,t] bind _ B (raise _ t) _ --> raise B t.

def try : A : prop ->
  tactic A -> (exception -> tactic A) -> tactic A.
[A,t] try A (ret _ t) _ --> ret A t
[t,f] try _ (raise _ t) f --> f t.
```

The tactic language

```
def intro_term : A : type ->
                 B : (term A -> prop) ->
                 (x : term A -> tactic (B x)) ->
                 tactic (all A B).
[A,B,b] intro_term A B (x => ret (B x) (b x))
      --> ret (all A B) (x : term A => b x)
[A,B,e] intro_term A B (x => raise (B x) e)
      --> raise (all A B) e.

def intro_proof : A : prop ->
                  B : prop ->
                  (proof A -> tactic B) ->
                  tactic (imp A B).
[A,B,b] intro_proof A B (x => ret B (b x))
      --> ret (imp A B) (x : proof A => b x)
[A,B,e] intro_proof A B (x => raise _ e)
      --> raise (imp A B) e.
```

Comparison with Mtac

This is a fragment of Mtac:

- FOL instead of CIC
- no direct manipulation of variables and meta-variables
- no encoding of fixpoint nor pattern-matching

But our tactic language is not as easy to use as Mtac because Dedukti lacks implicit arguments.

The certificate layer

- untyped
- concise
- Turing-complete

Certificates are meta-programs, they evaluate to tactics.
Certificate evaluation happens in a given context.

The certificate layer

```
context : Type.
nil_ctx : context.
cons_ctx_var :
  A : type -> term A -> context -> context.
cons_ctx_proof :
  A : prop -> proof A -> context -> context.

certificate : Type.

def run :
  A : prop -> context -> certificate -> tactic A.
```

The certificate layer

```
exact_mismatch : prop -> prop -> exception.
exact : A : prop -> proof A -> certificate.
[A,a] run A _ (exact A a) --> tactics.ret A a
[A,B] run A _ (exact B _)
       --> tactics.raise A (exact_mismatch A B).

raise : exception -> certificate.
[A,e] run A _ (raise e) --> tactics.raise A e.

try : certificate -> (exception -> certificate) ->
      certificate.
[A,G,c1,c2] run A G (try c1 c2) --> tactics.try ...
```

The certificate layer

```
with_goal : (prop -> certificate) -> certificate.
[A,G,c] run A G (with_goal c) --> run A G (c A).

with_context : (context -> certificate) -> certificate.
[A,G,c] run A G (with_context c) --> run A G (c G).

def with_assumption :
 (A : prop -> proof A -> certificate) -> certificate
:=
 f => with_context (G => try_all_assumptions f G).

def assumption : certificate := with_assumption exact.

clear : prop -> certificate -> certificate.
[A,G,B,c] run A G (clear B c) --> run A (ctx_remove B G)
```

The certificate layer

```
def match_prop : prop ->
certificate ->                                          (; false ;)
(prop -> prop -> certificate) ->                          (; and ;)
(prop -> prop -> certificate) ->                           (; or ;)
(prop -> prop -> certificate) ->                          (; imp ;)
(A:type -> (term A -> prop) -> certificate) ->  (; all ;)
(A:type -> (term A -> prop) -> certificate) ->   (; ex ;)
(p:dk_fol.predicate ->                                  (; pred ;)
 hlist (pred_arity p) term ->
 certificate) ->
certificate.
```

The certificate layer

```
[c]       match_prop false              c _ _ _ _ _ _ --> c
[A,B,c]   match_prop (and A B)          _ c _ _ _ _ _ --> c A B
[A,B,c]   match_prop (or A B)           _ _ c _ _ _ _ --> c A B
[A,B,c]   match_prop (imp A B)          _ _ _ c _ _ _ --> c A B
[A,B,c]   match_prop (all A B)          _ _ _ _ c _ _ --> c A B
[A,B,c]   match_prop (ex A B)           _ _ _ _ _ c _ --> c A B
[p,l,c]   match_prop (apply_pred p l)   _ _ _ _ _ _ c --> c p l.
```

The certificate layer

```
refine : A : prop -> B : prop ->
         (proof A -> proof B) ->
         certificate -> certificate.
[G,A,B,f,c] run B G (refine A B f c) -->
   tactics.bind A B (run A G c) (a : proof A =>
   tactics.ret B (f a)).

refine2 : A : prop -> B : prop -> C : prop ->
          (proof A -> proof B -> proof C) ->
          certificate -> certificate -> certificate.
[G,A,B,C,f,c1,c2] run C G (refine2 A B C f c1 c2) -->
    tactics.bind A C (run A G c1) (a : proof A =>
    tactics.bind B C (run B G c2) (b : proof B =>
    tactics.ret C (f a b))).
```

The certificate layer

```
intro : certificate -> certificate.
[A,B,G,c] run (dk_fol.imp A B) G (intro c)
      --> tactics.intro_proof A B ...
[A,B,G,c] run (dk_fol.all A B) G (intro c)
      --> tactics.intro_term A B ...

repeat : (certificate -> certificate) -> certificate.
[A,G,f] run A G (repeat f) --> run A G (f (repeat f)).
```

The certificate layer

```
def ifeq_term : A : type -> B : type ->
          term A -> term B.
[A,a] ifeq_term A A a --> a.

def ifeq_proof : A : prop -> B : prop ->
                 proof A -> proof B.
[A,a] ifeq_proof A A a --> a.
```

The certificate layer

```
def trivial : certificate.
def split : certificate -> certificate -> certificate.
def left : certificate -> certificate.
def right : certificate -> certificate.
def exists : A : type -> term A -> certificate ->
  certificate.

def modus_ponens : A : prop ->
  certificate -> certificate -> certificate.
def apply : A : type -> (term A -> prop) -> term A ->
  certificate -> certificate.
```

The certificate layer

```
def exfalso : certificate -> certificate.
def destruct_and : A : prop -> B : prop ->
  proof (and A B) -> certificate -> certificate.
def destruct_or : A : prop -> B : prop ->
  proof (or A B) -> certificate -> certificate ->
  certificate.
def destruct_imp : A : prop -> B : prop ->
  proof (imp A B) -> certificate -> certificate ->
  certificate.
def destruct_all : A : type -> B : (term A -> prop) ->
  proof (all A B) -> term A -> certificate ->
  certificate.
def destruct_ex :
  A : type -> B : (term A -> prop) -> proof (ex A B) ->
  certificate -> certificate.
```

Resolution

$$\frac{A \qquad B}{C}(\text{resolution})$$

if there are C_1, C_2, l, l', σ such that

- $A =_{AC} C_1 \vee l$,
- $B =_{AC} C_2 \vee \neg l'$,
- $C =_{AC} \sigma(C_1 \vee C_2)$,
- $\sigma(l) = \sigma(l')$.

Reasoning modulo AC

```
def modulo_ac_base : certificate :=
  certificates.repeat (t =>
    (certificates.try certificates.assumption
       (_ => certificates.try (certificates.left t)
          (_ => certificates.right t)))).

def modulo_ac : certificate :=
  certificates.repeat (mac =>
   certificates.with_assumption (A => a =>
    certificates.match_or A
     (A1 => A2 =>
      certificates.destruct_or A1 A2
        (certificates.ifeq_proof A (or A1 A2) a)
        (certificates.intro (certificates.clear A mac))
        (certificates.intro (certificates.clear A mac)))
     modulo_ac_base)).
```

Substitution

- Encoding of variables is shallow.
- A substitution is a list of pairs of terms.

$$
\begin{array}{lcl}
\sigma(f(t_1,\ldots,t_n)) & \hookrightarrow & f(\sigma(t_1),\ldots,\sigma(t_n)) \\
\{\}(t) & \hookrightarrow & t \\
\{x \mapsto a, \sigma\}(x) & \hookrightarrow & a \\
\{x \mapsto a, \sigma\}(t) & \hookrightarrow & \sigma(t)
\end{array}
$$

$$
\begin{array}{lcl}
\sigma(\bot) & \hookrightarrow & \bot \\
\sigma(A \square B) & \hookrightarrow & \sigma(A) \square \sigma(B) \\
\sigma(\mathcal{Q}x.A) & \hookrightarrow & \mathcal{Q}x.\sigma(A) \\
\sigma(P(t_1,\ldots,t_n)) & \hookrightarrow & P(\sigma(t_1),\ldots,\sigma(t_n)) \\
\{\}(A) & \hookrightarrow & A \\
\{x \mapsto x, \sigma\}(A) & \hookrightarrow & \sigma(A)
\end{array}
$$

Unification

- An equation is a pair of terms.
- A unification problem is a list of equations.
- A unification result is either the constant **FAIL** or a substitution

$$
\begin{array}{lcl}
O(x,x) & \hookrightarrow & \textbf{true} \\
O(x,f(t_1,\ldots,t_n)) & \hookrightarrow & O(x,t_1)\ \textbf{or}\ \ldots \textbf{or}\ O(x,t_n) \\
U([]) & \hookrightarrow & \{\} \\
U(f(t_1,\ldots,t_n)=f(t'_1,\ldots,t'_n)::p) & \hookrightarrow & U(t_1=t'_1::\ldots::t_n=t'_n::p) \\
U(f\_=g\_::\_) & \hookrightarrow & \textbf{FAIL} \\
U(t=t::p) & \hookrightarrow & U(p) \\
U(f(t_1,\ldots,t_n)=x::p) & \hookrightarrow & U(x=f(t_1,\ldots,t_n)::p) \\
U(x=t::p) & \hookrightarrow & UO(x,t,O(x,y),U(\{x\mapsto a\}(p))) \\
UO(\_,\_,\textbf{true},\_) & \hookrightarrow & \textbf{FAIL} \\
UO(\_,\_,\_,\textbf{FAIL}) & \hookrightarrow & \textbf{FAIL} \\
UO(x,t,\textbf{false},\sigma) & \hookrightarrow & \{x\mapsto\sigma(a),\sigma\}
\end{array}
$$

Simple resolution and specialization

$$\dfrac{\dfrac{\dfrac{A}{C_1 \vee l}\,\mathrm{AC}}{\sigma(C_1) \vee \sigma(l)}\,\mathrm{Specialize}(\sigma) \qquad \dfrac{\dfrac{B}{C_2 \vee \neg l'}\,\mathrm{AC}}{\sigma(C_2) \vee \neg\sigma(l')}\,\mathrm{Specialize}(\sigma)}{\dfrac{\sigma(C_1) \vee \sigma(C_2)}{C}\,\mathrm{AC}}\,\mathrm{Res}$$

Simple resolution and specialization

```
def simple_resolution (C1 : prop) (C2 : prop)
                      (l : prop)
                      (H1 : proof (or C1 l))
                      (H2 : proof (or C2 (not l)))
                      : proof (or C1 C2)
:= ...

def specialize (A : type)
               (B : term A -> prop)
               (f : term A -> term A)
               (H : proof (all A B))
               : proof (all A (x => B (f x)))
:= ...
```

Conclusion

- Rewriting is good for meta-programming
- Meta-programming is good for writing tactics
- Tactics are good for checking certificates

Contribution

- a typed tactic language and an untyped certificate language for Meta Dedukti
- expressive enough for certificate checking
- mostly independent of the object logic
  - but resolution relies on unification

Feature requests

- quoting
- meta-normalization
- local rewriting
- rewriting traces
- implicit arguments

Future work

applications to develop:

- certificate checking
- axiom elimination
- transfer

Next talk

- MathTransfer: a Dedukti library of transfer results generated from FoCaLiZe
- The `transfer` tactic in Meta Dedukti
- Zenon Modulo vs. `transfer` on MathTransfer