Tactics and certificates in Meta Dedukti[⋆]

Raphaël Cauderlier

University Paris Diderot, Irif, Paris, France

Abstract. Tactics are often featured in proof assistants to simplify the interactive development of proofs by allowing domain-specific automation. Moreover, tactics are also helpful to check the output of automatic theorem provers because they can rebuild details that the provers omit. We use meta-programming to define a tactic language for the Dedukti logical framework which can be used both for checking certificates produced by automatic provers and for developing proofs interactively.
More precisely, we propose a dependently-typed tactic language for first-order logic in Meta Dedukti and an untyped tactic language built on top of the typed one. We show the expressivity of these languages on two applications: a transfer tactic and a resolution certificate checker.

1 Introduction

Dedukti[23] is a logical framework implementing the $\lambda\Pi$-calculus modulo theories. It has been proposed as a universal proof checker[7]. In the tradition of the Edinburgh Logical Framework, Dedukti is based on the Curry-Howard isomorphism: it reduces the problem of checking proofs in an embedded logic to the problem of type-checking terms in a given signature. In order to express complex logical systems such as the Calculus of Inductive Constructions, Dedukti features rewriting: the user can declare rewrite rules handling the computational part of the system.

Proof translators from the proof assistants HOL Light, Coq, Matita, FoCaLiZe, and PVS to Dedukti have been developed and used to recheck proofs of these systems[1,2,10,16]. Moreover, Zenon Modulo[12] and iProver Modulo[9], two automatic theorem provers for an extension of classical first-order logic with rewriting known as Deduction modulo, are able to output proofs in Dedukti.

These proof-producing provers are helpful in the context of proof interoperability between proof assistants. Independently developed formal libraries often use equivalent but non identical definitions and these equivalences can often be proved by automatic theorem provers[11]. Hence the stronger proof automation in Dedukti is, the easiest it is to exchange a proof between proof assistants.

Dedukti is a mere type checker and it is intended to check machine-generated proofs, not to assist human users in the formalisation of mathematics. It lacks many features found in proof assistants to help the human user such as meta

⋆ This work has been supported in part by the VECOLIB project of the French national research organization ANR (grant ANR-14-CE28-0018).

variables, implicit arguments, and a tactic language. However these features, especially tactics implementing decision procedures for some fragments of the considered logic, can be very helpful to check less detailed proof *certificates* produced by automatic theorem provers and SMT solvers.

Fortunately, Dedukti already has all the features required to turn it into a powerful meta-programming language in which tactics and certificates can be transformed into proof objects. In this article, we propose a dependently typed monadic tactic language similar to Mtac[25]. This tactic language can be used for interactive proof development and certificate checking but because of the lack of implicit arguments in Dedukti, it is still very verbose. For this reason, we also introduce an untyped tactic language on top of the typed one to ease the writing of tactics.

Since our goal is to check certificates from automatic theorem provers and to construct proof object out of them, we focus in this article on the Dedukti encoding of classical first-order logic. In section 2, we present Dedukti and the encoding of classical first-order logic. The typed and untyped tactic languages are respectively presented in section 3 and section 4. Their applications to interactive proof development, theorem transfer, and certificate checking are shown in section 5, section 6, and section 7.

2 First-Order Logic in Dedukti

In this section, we present Dedukti by taking as example the encoding of first-order logic. We consider a multisorted first-order logic similar to the logics of the TPTP-TFF1[5] and SMTLIB[3] problem formats; its syntax of terms, and formulae is given in fig. 1. The logic is parameterized by a possibly infinite set of sorts $\mathcal{S}$. Each function symbol f has to be declared with a domain – a list of sorts $[A_1, \ldots, A_n]$ – and with a codomain $A \in \mathcal{S}$. A term of sort A is either a variable of sort A or a function symbol f of domain $[A_1, \ldots, A_n]$ and codomain A applied to terms $t_1, \ldots, t_n$ such that each t_i has sort A_i. Similarly, each predicate symbol P has to be declared with a domain $[A_1, \ldots, A_n]$. A formula is either an atom, that is a predicate symbol P of domain $[A_1, \ldots, A_n]$ applied to terms $t_1, \ldots, t_n$ such that each t_i has sort A_i or is obtained from the first-order logical connectives $\bot$ (falsehood), $\wedge$ (conjunction), $\vee$ (disjunction), $\Rightarrow$ (implication) and the quantifiers $\forall$ (universal) and $\exists$ (existential). As usual, we define negation by $\neg\varphi := \varphi \Rightarrow \bot$, truth by $\top := \neg\bot$, and equivalence by $\varphi_1 \Leftrightarrow \varphi_2 := (\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1)$.

$$\begin{array}{ll}
\textbf{Terms} & t := x \mid f(t_1, \ldots, t_n) \\
\textbf{Formulae} & \varphi := P(t_1, \ldots, t_n) \\
 & \quad \mid \ \bot \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2 \mid \forall x : A.\ \varphi \mid \exists x : A.\ \varphi
\end{array}$$

Fig. 1. Syntax of multisorted first-order logic

In Dedukti, we declare symbols for each syntactic class to represent: sorts, lists of sorts, terms, lists of terms, function symbols, predicate symbols, and formulae.

```
sort : Type.
sorts : Type.
term : sort -> Type.
terms : sorts -> Type.
function : Type.
predicate : Type.
prop : Type.
```

`Type` is Dedukti's builtin kind of types so the declaration `sort : Type.` means that `sort` is a Dedukti type and the declaration `term : sort -> Type.` means that `term` is a type family indexed by a sort.

Then we require domain and codomains for the symbols.

```
def fun_domain : function -> sorts.
def fun_codomain : function -> sort.
def pred_domain : predicate -> sorts.
```

The `def` keyword is used in Dedukti to indicate that the declared symbol is *definable*: this means that it is allowed to appear as head symbol in rewrite rules. In the case of the `fun_domain`, `fun_codomain`, and `pred_domain` functions, we do not give any rewrite rule now but each theory declaring new symbols is in charge of extending the definitions of these functions for the new symbols by adding the appropriate rewrite rules.

We then provide all the syntactic constructs, binding is represented using higher-order abstract syntax:

```
nil_sort : sorts.
cons_sort : sort -> sorts -> sorts.

nil_term : terms nil_sort.
cons_term : A : sort -> term A -> As : sorts -> terms As ->
  terms (cons_sort A As).

fun_apply : f : function -> terms (fun_domain f) -> term (fun_codomain f).

pred_apply : p : predicate -> terms (pred_domain p) -> prop.
false : prop.
and : prop -> prop -> prop.
or : prop -> prop -> prop.
imp : prop -> prop -> prop.
all : A : sort -> (term A -> prop) -> prop.
ex : A : sort -> (term A -> prop) -> prop.

def not (a : prop) := imp a false.
def eqv (a : prop) (b : prop) := and (imp a b) (imp b a).
```

The types of `cons_term`, `fun_apply`, `pred_apply`, `all`, and `ex` use the dependent product $\Pi x : A.\ B$ where x might occur in B; it is written `x : A -> B` in Dedukti.

Finally, we define what it means to be a proof of some proposition. For this we could declare symbols corresponding to the derivation rules of some proof system such as natural deduction or sequent calculus. However, the standard way to do this for first-order logic in Dedukti is to use the second-order definition of connectives and then derive the rules of natural deduction.

```
def proof : prop -> Type.
[] proof false --> a : prop -> proof a
[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) --> x : term A -> proof (p x)
[A,p] proof (ex A p) -->
  c: prop -> (x : term A -> proof (p x) -> proof c) -> proof c.
```

Each rewrite rule in this definition of `proof` has the form `[context] lhs --> rhs`. The context lists the free variables appearing in the left-hand side, the left-hand side is a pattern (a first-order pattern in this case but higher-order patterns in the sense of Miller[21] are also supported by Dedukti) and the right-hand side is a term whose free variables are contained in the context.

All the rules of natural deduction can now be proved, here is for example, the introduction rule for conjunction:

```
def and_intro (a : prop) (b : prop) (Ha : proof a) (Hb : proof b)
  : proof (and a b) :=
  c : prop => f : (proof a -> proof b -> proof c) => f Ha Hb.
```

The syntax `x : A => b` is used in Dedukti for the λ-abstraction $\lambda x : A.\ b$.

To check the certificates found by automatic theorem provers for classical logic, we need two axiom schemes: the law of excluded middle and the assumption that all sorts are inhabited.

```
excluded_middle : a : prop -> proof (or a (not a)).
default_value : A : sort -> term A.
```

The Dedukti signature that we have described in this section is a faithful encoding of classical first-order logic[14]: a first-order formula φ is provable in classical natural deduction if and only if the Dedukti type `proof` φ is inhabited.

3 A Typed Tactic Language for Meta Dedukti

Unfortunately, writing Dedukti terms in the signature of the previous section is tedious not only for human users but also for automated tools which typically reason at a higher level than natural deduction proofs. In this section, we propose a first tactic language to ease the creation of terms in this signature.

Since Dedukti does not check for termination, it is very easy to encode a Turing-complete language in Dedukti. For example, the untyped λ-calculus can be encoded with only one declaration `def A : Type.` and one rewrite rule `[ ] A --> A -> A.`

Thanks to Turing-completeness, we can use Dedukti as a dependently-typed programming language based on rewriting. The results of these programs are Dedukti terms that need to be checked in a trusted Dedukti signature such as the one of section 2 if we want to interpret them as proofs. We distinguish two different Dedukti signatures: the trusted signature of section 2 and an untrusted signature extending the one of section 2 and used to elaborate terms to be checked in the trusted one. Unless otherwise precised, all the Dedukti excerpts from now on are part of this second, untrusted signature.

When using Dedukti as a meta-programming language, we are not so much interested in the type-checking problem than in the normal forms (with respect to the untrusted system) of some terms. For this reason, we use a fork of Dedukti called Meta Dedukti[13] that we developed with Thiré. This tool outputs a copy of its input Dedukti file in which each term is replaced by its normal form. The produced file can then be sent to Dedukti to be checked in the trusted signature.

```
exc : Type.
mtactic : prop -> Type.

mret : A : prop -> proof A -> mtactic A.
mraise : A : prop -> exc -> mtactic A.
def mrun : A : prop -> mtactic A -> proof A.
def mbind : A : prop -> B : prop ->
  mtactic A -> (proof A -> mtactic B) -> mtactic B.
def mtry : A : prop -> mtactic A -> (exc -> mtactic A) -> mtactic A.
def mintro_term : A : sort -> B : (term A -> prop) ->
                  (x : term A -> mtactic (B x)) -> mtactic (all A B).
def mintro_proof : A : prop -> B : prop ->
                   (proof A -> mtactic B) -> mtactic (imp A B).
```

Fig. 2. The typed tactic language: declarations

In fig. 2 and fig. 3 we define our typed tactic language for Meta Dedukti inspired by the MTac tactic language for Coq[25]. The main type of this development is the type `mtactic a` (for <u>m</u>onadic <u>tactic</u>) where `a` is a proposition. We call *tactical* any function returning a term of type `mtactic a` for some `a`. A term `t` of type `mtactic a` contains instructions to attempt a proof of the proposition `a`. Each tactic can either fail, in which case its normal form is `mraise a e` where `e` is of type `exc`, an extensible type of exceptions or succeed in which case its normal form is `mret a p` where `p` is a proof of `a`. The tacticals `mret` and `mraise` can be seen as the two constructors of the inductive type family `mtactic`. When evaluating a tactic is successful, we can extract the produced proof using the `mrun` partial function which is undefined in the case of the `mraise` constructor. Tactics can be chained using the `mbind` tactical and backtracking points can be set using the `mtry` tactical.

```
[a] mrun _ (mret _ a) --> a.

[f,t] mbind _ _ (mret _ t) f --> f t
[B,t] mbind _ B (mraise _ t) _ --> mraise B t.

[A,t] mtry A (mret _ t) _ --> mret A t
[t,f] mtry _ (mraise _ t) f --> f t.

[A,B,b] mintro_term A B (x => mret (B x) (b x)) -->
   mret (all A B) (all_intro A B (x => b x))
[A,B,e] mintro_term A B (x => mraise (B x) e) --> mraise (all A B) e.

[A,B,b] mintro_proof A B (x => mret B (b x)) -->
   mret (imp A B) (imp_intro A B (x => b x))
[A,B,e] mintro_proof A B (x => mraise _ e) --> mraise (imp A B) e.
```

Fig. 3. The typed tactic language: rewrite rules

The `mbind` tactical is enough to define tactics corresponding to all the rules of natural deduction that do not change the proof context. As a simple example, we can define a `msplit` tactical attempting to prove goals of the form `and a b` from tactics `t1` and `t2` attempting to prove `a` and `b` respectively.

```
def msplit (a : prop) (b : prop) (t1 : mtactic a) (t2 : mtactic b)
  : mtactic (and a b) :=
  mbind a (and a b) t1 (Ha =>
  mbind b (and a b) t2 (Hb =>
  mret (and a b) (and_intro a b Ha Hb))).
```

To handle the natural deduction rules that do modify the rule context such as the introduction rules for implication and universal quantification, we add two new tacticals `mintro_term` and `mintro_proof`. These tacticals are partial functions only defined if their argument is a tactical that uniformly succeed on all arguments or uniformly fail on all arguments.

4 An Untyped Tactic Language for Meta Dedukti

The main limitation of the typed tactic language presented in section 3 is its verbosity. Since Dedukti does not feature implicit arguments, each time the user applies the `msplit` tactical, she has to provide propositions `a` and `b` such that the goal to be proved is convertible with `and a b`. Another issue is that this tactic language does not permit to automate search among assumptions; new assumptions can be introduced by the `mintro_proof` tactical but the user of the typed tactic language then has to refer explicitly to the introduced assumption.

The untyped[1] tactic language that we now consider solves both issues. Tactics are interpreted in a proof context, a list of terms and proofs, by the `eval` function returning a typed tactic. For the common case of evaluating a tactic in the empty context, we define the `prove` function.

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

tactic : Type.
def eval : context -> goal : prop -> tactic -> mtactic goal.

def prove (a : prop) (t : tactic) : proof a :=
  mrun a (eval nil_ctx a t).
```

Some of the most fundamental tacticals of the untyped language are defined in fig. 4 by the way `eval` behaves on them. The `with_goal` tactical is used to get access to the current goal, it takes another tactical as argument and evaluates it on the goal. The `with_assumption` tactical tries a tactical on each assumption of the context until one succeeds. The `exact`, `raise`, `try`, `bind` and `intro` tacticals are wrapper around the constructs of the typed language. The full definitions of these tacticals and many other are available in the file `https://gitlab.math.univ-paris-diderot.fr/cauderlier/dktactics/blob/master/meta/tactic.dk`

On top of these basic tacticals, we have implemented tacticals corresponding to the rules of intuitionistic sequent calculus. For example, fig. 5 presents the definitions of the tacticals about conjunction: `match_and` deconstructs formulae of the form `and a b`, `split` performs the right rule of conjunction in sequent calculus and is defined very similarly to `msplit`, its typed variant of section 3. The tactical `destruct_and` implements the following generalisation of the left rule for conjunction:

$$\frac{\Gamma \vdash A \wedge B \qquad \Gamma, A, B \vdash C}{\Gamma \vdash C}$$

The axiom rule of sequent calculus is implemented by the `assumption` tactic defined as `with_assumption exact`.

[1] By "untyped" we do not mean that no type is assigned to the Dedukti terms of the language but that typing is trivial: all the tactics have the same type (`tactic`).

```
with_goal : (prop -> tactic) -> tactic.
[ctx,goal,F] eval ctx goal (with_goal F) --> eval ctx goal (F goal).

with_assumption : (A : prop -> proof A -> tactic) -> tactic.
[ctx,goal,F] eval ctx goal (with_assumption F) --> ...

exact_mismatch : exc.
exact : a : prop -> proof a -> tactic.
[a,H] eval _ a (exact a H) --> mret a H
[a] eval _ a (exact _ _) --> mraise a exact_mismatch.

raise : exc -> tactic.
[a,e] eval _ a (raise e) --> mraise a e.

try : tactic -> (exc -> tactic) -> tactic.
[ctx,goal,t,f] eval ctx goal (try t f) --> mtry ...

bind : A : prop -> tactic -> (proof A -> tactic) -> tactic.
[ctx,goal,A,t,f] eval ctx goal (bind A t f) --> mbind ...

intro_failure : exc.
intro : tactic -> tactic.
[ctx,a,b,t] eval ctx (imp a b) (intro t) --> mintro_proof ...
[ctx,A,p,t] eval ctx (all A p) (intro t) --> mintro_term ...
[goal] eval _ goal (intro _) --> mraise goal intro_failure.
```

Fig. 4. Low-level untyped tacticals

```
matching_failure : exc.

def match_and : prop -> (prop -> prop -> tactic) -> tactic.
[a,b,t] match_and (and a b) t --> t a b
[] match_and _ _ --> raise matching_failure.

def split (t1 : tactic) (t2 : tactic) :=
  with_goal (goal => match_and goal
    (a => b => bind a t1 (Ha =>
               bind b t2 (Hb =>
               exact (and a b) (and_intro a b Ha Hb))))).

def destruct_and (a : prop) (b : prop) (tab : tactic) (t : tactic) :=
  with_goal (goal => bind (and a b) tab (Hab =>
    bind (imp a (imp b goal)) (intro (intro t)) (Hf =>
    exact goal (Hf (and_elim_1 a b Hab) (and_elim_2 a b Hab))))).
```

Fig. 5. Conjunction tacticals

5 Example of Interactive Proof Development

Before considering sophisticated applications of our tactic languages in section 6 and section 7, we illustrate the interactive use of our untyped tactic language on a simple example: commutativity of conjunction.

We start with the following Dedukti file:

```
def t0 : tactic.

def and_commutes (a : prop) (b : prop) : proof (imp (and a b) (and b a))
:= prove (imp (and a b) (and b a)) t0.
```

The undefined constant `t0` is a placeholder for an unsolved goal. The interactive process consists in looking into the normal form of this file for blocked applications of the `eval` function, adding some lines after the declaration of `t0`, and repeating until the definition of `and_commutes` is a term of the encoding of section 2.

At the first iteration, Meta Dedukti answers

```
def t0 : tactic.

def and_commutes : a:prop -> b:prop ->
  (c:prop -> ((proof a) -> (proof b) -> proof c) -> proof c) ->
  c:prop -> ((proof b) -> (proof a) -> proof c) -> proof c
  := a:prop => b:prop =>
  mrun (imp (and a b) (and b a))
    (eval nil_ctx (imp (and a b) (and b a)) t0).
```

We have one blocked call to `eval` on the last line: `eval nil_ctx (imp (and a b) (and b a)) t0`; this means we have to prove $\vdash (a \wedge b) \Rightarrow (b \wedge a)$. To apply the `intro` tactical, we introduce a new undefined subgoal `t1` and define `t0` as `intro t1` by adding the following line in the middle of our file.

```
def t1 : tactic.  [ ] t0 --> intro t1.
```

Normalising again produces a file containing the term `eval (cons_ctx_proof (and a b) a0 nil_ctx) (and b a) t1` which means we now have to prove $a \wedge b \vdash b \wedge a$. To do this we add the following lines right after the previously added line:

```
def t2 : tactic.
def t3 : tactic.
[t1] t1 --> with_assumption (c => H => match_and c
    (a => b => destruct_and a b t2 t3)).
```

In other words we try to apply the `destruct_and` tactical successively to all assumptions of the proof context. Since we have only one assumption and it is indeed a conjunction, the call reduces and Meta Dedukti tells us that we are left with `eval (cons_ctx_proof (and a b) a0 nil_ctx) (and a b) t2` and `eval (cons_ctx_proof b a3 (cons_ctx_proof a a2 (cons_ctx_proof (and a b) a0 nil_ctx))) (and b a) t3` corresponding respectively to $a \wedge b \vdash a \wedge b$ and $b, a, a \wedge b \vdash b \wedge a$. The first subgoal is trivial and can be solved with the `assumption` tactic that succeeds when the goal matches one of the assumptions. For the second subgoal, we introduce the conjunction.

```
[] t2 --> assumption.
def t4 : tactic.
def t5 : tactic.
[] t3 --> split t4 t5.
```

We again have two subgoals, `eval (cons_ctx_proof b a2 (cons_ctx_proof a a1 (cons_ctx_proof (and a b) a0 nil_ctx))) b t4` and `eval (cons_ctx_proof b a2 (cons_ctx_proof a a1 (cons_ctx_proof (and a b) a0 nil_ctx))) a t5` corresponding to $b, a, a \wedge b \vdash b$ and $b, a, a \wedge b \vdash a$. In both cases, the goal corresponds to one of the assumptions so the `assumption` tactic does the job.

```
[] t4 --> assumption.
[] t5 --> assumption.
```

Our theorem is now proved; the following definition of `and_commutes` given by Meta Dedukti is accepted by Dedukti:

```
def and_commutes : a:prop -> b:prop ->
  (c:prop -> ((proof a) -> (proof b) -> proof c) -> proof c) ->
  c:prop -> ((proof b) -> (proof a) -> proof c) -> proof c
  :=
  a:prop => b:prop => x => c:prop =>
  f:((proof b) -> (proof a) -> proof c) =>
  f (x b (x0 => y => y)) (x a (x0 => y => x0)).
```

6 Theorem Transfer

When translating independently developed formal libraries in Dedukti, we end up with two isomorphic copies A and B of the same notions. Contrary to the mathematical habit of identifying isomorphic structures, in formal proof systems a theorem φ_A on the structure A cannot be used without justification as a theorem φ_B on the structure B. However this justification, a proof of $\varphi_A \Rightarrow \varphi_B$, can be automated in tactic based proof assistants. The automation of such goals of the form $\varphi_A \Rightarrow \varphi_B$ is called theorem transfer[17,26] and the tactic implementing it is called a transfer tactic.

In fig. 6, we adapt the higher-order transfer calculi of [17] and [26] to first-order logic. The notations $P\ (\mathcal{R}_1 \times \ldots \times \mathcal{R}_n)\ Q$ abbreviates the formula $\forall x_1, \ldots x_n, y_1, \ldots y_n.\ x_1\ \mathcal{R}_1\ y_1 \Rightarrow \ldots \Rightarrow x_n\ \mathcal{R}_n\ y_n \Rightarrow P(x_1, \ldots, x_n) \Rightarrow Q(y_1, \ldots, y_n)$ and the notation $f\ (\mathcal{R}_1 \times \ldots \times \mathcal{R}_n \to \mathcal{R})\ g$ abbreviates the formula $\forall x_1, \ldots x_n, y_1, \ldots y_n.\ x_1\ \mathcal{R}_1\ y_1 \Rightarrow \ldots \Rightarrow x_n\ \mathcal{R}_n\ y_n \Rightarrow f(x_1, \ldots, x_n)\ \mathcal{R}\ g(y_1, \ldots, y_n)$.

$$\frac{}{\Gamma \vdash \bot \Rightarrow \bot} \qquad \frac{\Gamma \vdash \varphi_1 \Rightarrow \psi_1 \qquad \Gamma \vdash \varphi_2 \Rightarrow \psi_2}{\Gamma \vdash (\varphi_1 \wedge \varphi_2) \Rightarrow (\psi_1 \wedge \psi_2)}$$

$$\frac{\Gamma \vdash \varphi_1 \Rightarrow \psi_1 \qquad \Gamma \vdash \varphi_2 \Rightarrow \psi_2}{\Gamma \vdash (\varphi_1 \vee \varphi_2) \Rightarrow (\psi_1 \vee \psi_2)} \qquad \frac{\Gamma \vdash \psi_1 \Rightarrow \varphi_1 \qquad \Gamma \vdash \varphi_2 \Rightarrow \psi_2}{\Gamma \vdash (\varphi_1 \Rightarrow \varphi_2) \Rightarrow (\psi_1 \Rightarrow \psi_2)}$$

$$\frac{\Gamma, a : A, c : C, a\ \mathcal{R}\ c \vdash \varphi_a \Rightarrow \psi_c \qquad \vdash \forall c : C.\ \exists a : A.\ a\ \mathcal{R}\ c}{\Gamma \vdash (\forall a : A.\ \varphi_a) \Rightarrow (\forall c : C.\ \psi_c)}$$

$$\frac{\Gamma, a : A, c : C, a\ \mathcal{R}\ c \vdash \varphi_a \Rightarrow \psi_c \qquad \vdash \forall a : A.\ \exists c : C.\ a\ \mathcal{R}\ c}{\Gamma \vdash (\exists a : A.\ \varphi_a) \Rightarrow (\exists c : C.\ \psi_c)}$$

$$\frac{\Gamma \vdash t_1\ \mathcal{R}_1\ u_1 \quad \ldots \quad \Gamma \vdash t_n\ \mathcal{R}_n\ u_n \qquad \vdash P\ (\mathcal{R}_1 \times \ldots \times \mathcal{R}_n)\ Q}{\Gamma \vdash P(t_1, \ldots, t_n) \Rightarrow Q(u_1, \ldots, u_n)}$$

$$\frac{\Gamma \vdash t_1\ \mathcal{R}_1\ u_1 \quad \ldots \quad \Gamma \vdash t_n\ \mathcal{R}_n\ u_n \qquad \vdash f\ (\mathcal{R}_1 \times \ldots \times \mathcal{R}_n \to \mathcal{R})\ g}{\Gamma \vdash f(t_1, \ldots, t_n)\ \mathcal{R}\ g(u_1, \ldots, u_n)}$$

Fig. 6. A first-order transfer calculus

Implementing a proof search algorithm for this calculus in our untyped tactic language is straightforward once we have proved the formula schemes $\bot \Rightarrow \bot$, $(\varphi_1 \Rightarrow \psi_1) \Rightarrow (\varphi_2 \Rightarrow \psi_2) \Rightarrow (\varphi_1 \wedge \varphi_2) \Rightarrow (\psi_1 \wedge \psi_2)$, $(\varphi_1 \Rightarrow \psi_1) \Rightarrow (\varphi_2 \Rightarrow \psi_2) \Rightarrow (\varphi_1 \vee \varphi_2) \Rightarrow (\psi_1 \vee \psi_2)$, ... corresponding to the rules of the calculus.

Instead of deriving the proofs of these formula schemes in natural deduction directly, we take benefit of our tactic language to define an `auto` tactic following a rather naive strategy for sequent calculus: it applies right rules for all connectives but the existential quantifier as long as possible and then applies left rules for all connectives but universal quantification until the goal matches one of the assumptions. The auto tactic is able to prove the four first rules of our transfer calculus. The four remaining rules require to instantiate universal assumptions and are hence out of its scope but they are easy to prove directly.

Our implementation is available at the following URL: `https://gitlab.math.univ-paris-diderot.fr/cauderlier/dktransfer`.

7 Resolution Certificates

Robinson's resolution calculus[22] is a popular proof calculus for first-order automatic theorem provers. It is a clausal calculus; this means that it does not handle the full syntax of first-order formulae but only the CNF (clausal normal form) fragment.

A *literal* is either an atom (a positive literal) or the negation of an atom (a negative literal). We denote by $\bar{l}$ the opposite literal of l defined by $\bar{a} := \neg a$

and $\overline{\neg a} := a$ where a is any atom. A *clause* is a possibly empty disjunction of literals. The empty clause corresponds to falsehood. Literals and clauses may contain free variables which are to be interpreted as universally quantified. We make this explicit by considering *quantified clauses* (qclauses for short) which are formulae of the form $\forall x_1, \dots, x_k.\ l_1 \vee \dots \vee l_n$.

A resolution proof is a derivation of the empty clause from a set of clauses assumed as axioms. The rules of the resolution calculus are given in fig. 7. The (*Factorisation*) and (*Resolution*) rules are standard, the (*Unquantification*) rule is required to remove useless quantifications in the clauses produced by the two other rules. Note that the correctness of this (*Unquantification*) rule requires the `default_value` axiom that we introduced in section 2.

$$\frac{\forall \overrightarrow{x}.\ l_1 \vee \dots \vee l_n \qquad \sigma = mgu(l_i, l_j)}{\forall \overrightarrow{x}.\ \sigma(l_1 \vee \dots \vee l_{j-1} \vee l_{j+1} \vee \dots \vee l_n)}(\mathit{Factorisation})$$

$$\frac{\forall \overrightarrow{x}.\ l_1 \vee \dots \vee l_n \qquad \forall \overrightarrow{y}.\ l'_1 \vee \dots \vee l'_m \qquad \sigma = mgu(l_i, \overline{l_j})}{\forall \overrightarrow{x}, \overrightarrow{y}.\ \sigma(l_1 \dots l_{i-1} \vee l_{i+1} \dots l_n \vee l'_1 \dots l'_{j-1} \vee l'_{j+1} \dots l'_m)}(\mathit{Resolution})$$

$$\frac{\forall \overrightarrow{x}.\ C \qquad FV(C) = \overrightarrow{y}}{\forall \overrightarrow{y}.\ C}(\mathit{Unquantification})$$

Fig. 7. The resolution calculus with quantified clauses

We consider resolution certificates in which the assumed and derived clauses are numbered and each line of the certificate indicates:

1. the name of the derivation rule (either "Factorisation" or "Resolution"),
2. the numbers identifying one or two (depending on the chosen derivation rule) previously assumed or derived clauses,
3. the indexes i and j of the literals to unify, and
4. the number of the newly derived clause.

This level of detail is not unreasonable to ask from a resolution prover; Prover9[20] for example is able to produce such certificates. To express these certificates in Meta Dedukti, we have extended the trusted signature of first-order logic with the definitions of the syntactic notions of atoms, literals, clauses, and qclauses (see fig. 8) and we have defined functions `factor`, `resolve`, and `unquantify` returning the qclause resulting respectively from the (*Factorisation*), (*Resolution*), and (*Unquantification*) rules and tacticals `factor_correct`, `resolve_correct`, and `unquantify_correct` attempting to prove the resulting clauses from proofs of the initial clauses. Moreover, we defined a partial function `qclause_of_prop` mapping propositions in the clausal fragment to quantified clauses and we proved it correct on this fragment. The signature of these functions is given in fig. 9.

```
atom : Type.
mk_atom : p : predicate -> terms (pred_domain p) -> atom.

literal : Type.
pos_lit : atom -> literal.
neg_lit : atom -> literal.

clause : Type.
empty_clause : clause.
cons_clause : literal -> clause -> clause.

qclause : Type.
qc_base : clause -> qclause.
qc_all : A : sort -> (term A -> qclause) -> qclause.
```

Fig. 8. Syntactic definitions for the CNF fragment of first-order logic

```
def cprop : qclause -> prop.

def qclause_of_prop : prop -> qclause.
def qclause_of_prop_correct : a : prop -> proof a ->
  mtactic (cprop (qclause_of_prop a)).

def factor : nat -> nat -> qclause -> qclause.
def resolve : nat -> nat -> qclause -> qclause -> qclause.

def factor_correct : i : nat -> j : nat -> C : qclause ->
  proof (cprop C) -> mtactic (cprop (factor i j C)).
def resolve_correct : i : nat -> j : nat -> C : qclause -> D : qclause ->
  proof (cprop C) -> proof (cprop D) -> mtactic (cprop (resolve i j C D)).

def unquantify : qclause -> qclause.
def unquantify_correct : C : qclause -> proof (cprop C) ->
  mtactic (cprop (unquantify C)).
```

Fig. 9. Signature of the resolution tacticals

As a small example illustrating the use of these tacticals, we consider the problem `NUM343+1` from the TPTP benchmark[24]. Among the clauses resulting from the clausification of the problem, two of them are used in the proof found by Prover9: $x \leq y \vee y \leq x$ and $\neg(x \leq n)$. The translation of this problem in Dedukti is given in fig. 10. Here is a resolution certificate of the empty clause from these axioms:

1.	$x \leq y \vee y \leq x$	Axiom
2.	$\neg(x \leq n)$	Axiom
3.	$x \leq x$	Factorisation at positions 0 and 1 in clause 1
4.	$\bot$	Resolution at positions 0 and 0 in clauses 2 and 3

```
(; Signature ;)
A : sort.
LEQ : predicate.
[] pred_domain LEQ --> cons_sort A (cons_sort A nil_sort).
N : function.
[] fun_domain N --> nil_sort.
[] fun_codomain N --> A.
def leq (a : term A) (b : term A) : prop :=
  pred_apply LEQ
    (cons_term A a (cons_sort A nil_sort)
      (cons_term A b nil_sort nil_term)).
def n : term A := fun_apply N nil_term.

(; Axioms ;)
def A0 := all A (x => all A (y => or (leq x y) (leq y x))).
a0 : proof A0.
def A1 := all A (x => not (leq x n)).
a1 : proof A1.
```

Fig. 10. The TPTP problem `NUM343+1` in Meta Dedukti

This certificate can be translated in our formalism by adding an (*Unquantification*) step after each other step. The Meta Dedukti version of this certificate is given in fig. 11, once normalized by Meta Dedukti, we obtain a Dedukti file of 518 lines that is successfully checked by Dedukti in the trusted signature.

During the definition of the tacticals of fig. 9, we were happily surprised to discover that the tacticals `qclause_of_prop_correct`, `factor_correct`, `resolve_correct`, and `unquantify_correct` were not much harder to define than the corresponding clause computing functions because we did not prove the soundness of the resolution calculus. In particular, we did not prove the correctness of our unification algorithm but we check *a posteriori* that the returned substitution is indeed an unifier of the given literals. The main difficulty comes

```
(; Clauses ;)
def C0 := qclause_of_prop A0.
def c0 := mrun (cprop C0) (qclause_of_prop_correct A0 a0).
def C1 := qclause_of_prop A1.
def c1 := mrun (cprop C1) (qclause_of_prop_correct A1 a1).
def C2' := factor 0 1 C0.
def c2' := mrun (cprop C2') (factor_correct 0 1 C0 c0).
def C2 := unquantify C2'.
def c2 := mrun (cprop C2) (unquantify_correct C2' c2').
def C3' := resolve 0 0 C1 C2.
def c3' := mrun (cprop C3') (resolve_correct 0 0 C1 C2 c1 c2).
def C3 := unquantify C3'.
def c3 : proof false := mrun (cprop C3) (unquantify_correct C3' c3').
```

Fig. 11. A resolution certificate for TPTP problem `NUM343+1` in Meta Dedukti

from the application of substitution to qclauses which can be isolated in a rule called specialisation:

$$\frac{\forall x_1.\ \dots \forall x_n.\ C}{\forall x_1.\ \dots \forall x_n.\ \sigma(C)}\,(\mathit{Specialisation})$$

If `c` is a proof of $\forall \overrightarrow{x}.\ C$, then a proof of $\forall \overrightarrow{x}.\ \sigma(C)$ can be obtained by first introducing all the quantifiers then applying `c` to $\sigma(x_1), \dots, \sigma(x_n)$. From our tactic languages, it is not easy to do this because the number n of introduction and elimination rules to apply is unknown. To solve this problem, we defined an alternative form of quantified clauses were instead of quantifying over terms one by one, we quantify over lists of terms: `def lqclause (As : sorts) : Type := terms As -> clause`. We proved the specialisation rule on this type `lqclause` and the equivalence between `lqclause` and `qclause`.

The tactic languages of section 3 and section 4 and the resolution certificate checker of this section are available at the following URL: `https://gitlab.math.univ-paris-diderot.fr/cauderlier/dktactics`.

8 Related Works

The main source of inspiration for the typed tactic language that we have proposed in section 3 is MTac[25], a typed monadic language for the Coq proof assistant. Our language is a fragment of MTac; the missing MTac primitives provide non-termination (the **mfix** construct) and give access to operations of Coq refiner such as syntactic deconstruction of terms (**mmatch**), higher-order unification, and handling of meta variables. To provide these features, the operational semantics of MTac is implemented inside Coq refiner. In this work in contrast, we did not modify Dedukti at all. The **mfix** and **mmatch** operations are not needed in our tactic languages because the user already has access to

Dedukti higher-order rewrite engine. Since Dedukti is not a full proof assistant but only a proof checker, it does not currently feature a refiner from which we could leverage higher-order unification or meta variables. However, as we have seen in section 5, we can simulate meta variables by definable symbols of type `tactic` and as we have seen in section 7 in the first-order case we can also define the unification algorithm.

A second version of MTac is in preparation[19]. In MTac2, an untyped tactic language is built on top of the MTac monad but contrary to our untyped language in which tactics promise proofs of the current goal, MTac2 tactics promise lists of subgoals and the actual proof is built by instanciation of meta variables. This gives MTac2 the flexibility to define tactics generating a number of subgoals that is not known statically.

Exceptions and backtracking are also implemented by a monad in the meta language of Lean which is used to implement Lean tactics[15]. However, Lean meta language is poorly typed making this tactic language closer to our untyped tactic language: the way tactics manipulate the proof state in Lean is not made explicit in their types and terms are all reified in the same type `expr`.

The tactics of the Idris[8] system, which are used to elaborate terms from the full Idris syntax to the syntax of Idris' kernel, are also implemented by a monad in Haskell. However, this tactic monad is not reflected in Idris so Idris users do not have access to an extensible tactic language.

To bridge the gap between automatic and interactive theorem proving, a lot of efforts has been put to check the certificates of automatic theorem provers. iProver Modulo[9], Zenon Modulo[12], and Metis[18] are first-order theorem provers able to produce independently checkable proofs. Metis in particular can be used as a tactic in Isabelle/HOL. The Sledgehammer tool[4] checks certificates from first-order provers and SMT solvers using Isabelle tactics implementing decision procedures and the Metis tactic. These works have in common an access to a deep representation of terms, typically using De Bruijn indices or named variables, at proof producing time whereas our tactics for the resolution calculus only use higher-order abstract syntax. Recently, the Foundational Proof Certificate framework has been used to add enough details to Prover9 resolution certificates that they can be checked by a simple tool that does not need to compute the unifiers.[6] In our context, we have found that is was actually easier to perform the unification in the certificate checker than to extend the format of certificates to include the substitutions because the naming of free variables in clauses (or the order in which variables are implicitly quantified) is hard to predict.

9 Conclusion

We have shown that Dedukti could be used as an expressive meta language for writing tactics and checking proof certificates. We have proposed two tactic languages for Dedukti, a typed one and an untyped one and shown applications of these languages to interactive proof development, automated theorem transfer, and checking of resolution certificates.

For interactive proof development and tactic debugging, our languages would greatly benefit from pretty-printing functions. We believe such functions can be defined in a second meta signature used to transform blocked `eval` calls to something more readable.

Our tactic and certificate languages are defined specifically for first-order logic. Since it was inspired by tactic languages for the Calculus of Inductive Constructions, we believe that most of the work presented in this article can be adapted straightforwardly to richer logics with the notable exception of the unification algorithm used to check resolution certificates.

Most clausal first-order theorem provers use an extra rule called paramodulation to handle equality. We would like to extend our resolution certificate language to take this rule into account. This would allow us to benchmark our certificate checker on large problem sets such as TPTP.

References

1. Assaf, A.: A Framework for Defining Computational Higher-Order Logics. Ph.D. thesis, École Polytechnique (2015), `https://tel.archives-ouvertes.fr/tel-01235303`
2. Assaf, A., Burel, G.: Translating HOL to Dedukti. In: Kaliszyk, C., Paskevich, A. (eds.) Proceedings Fourth Workshop on Proof eXchange for Theorem Proving, Berlin, Germany, August 2-3, 2015. Electronic Proceedings in Theoretical Computer Science, vol. 186, pp. 74–88. Open Publishing Association, Berlin, Germany (August 2015). https://doi.org/10.4204/EPTCS.186.8
3. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2016), `http://smtlib.cs.uiowa.edu`
4. Blanchette, J.C., Bulwahn, L., Nipkow, T.: Automatic Proof and Disproof in Isabelle/HOL. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) Frontiers of Combining Systems, 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6989, pp. 12–27. Springer (2011). https://doi.org/10.1007/978-3-642-24364-6_2
5. Blanchette, J.C., Paskevich, A.: TFF1: the TPTP typed first-order form with rank-1 polymorphism. In: Bonacina, M.P. (ed.) Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. LNCS, vol. 7898, pp. 414–420. Springer (2013). https://doi.org/10.1007/978-3-642-38574-2_29
6. Blanco, R., Chihani, Z., Miller, D.: Translating between implicit and explicit versions of proof. In: de Moura, L. (ed.) Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10395, pp. 255–273. Springer (2017). https://doi.org/10.1007/978-3-319-63046-5_16
7. Boespflug, M., Carbonneaux, Q., Hermant, O.: The $\lambda\Pi$-calculus Modulo as a Universal Proof Language. In: David Pichardie, T.W. (ed.) the Second International Workshop on Proof Exchange for Theorem Proving (PxTP 2012). vol. Vol. 878, pp. pp. 28–43. Manchester, United Kingdom (Jun 2012), `https://hal-mines-paristech.archives-ouvertes.fr/hal-00917845`

8. Brady, E.: Idris, a general-purpose dependently typed programming language: Design and implementation. J. Funct. Program. **23**(5), 552–593 (2013). https://doi.org/10.1017/S095679681300018X
9. Burel, G.: A shallow embedding of resolution and superposition proofs into the $\lambda\Pi$-calculus modulo. In: Blanchette, J.C., Urban, J. (eds.) Third International Workshop on Proof Exchange for Theorem Proving, PxTP 2013, Lake Placid, NY, USA, June 9-10, 2013. EPiC Series in Computing, vol. 14, pp. 43–57. EasyChair, Lake Placid, USA (June 2013), http://www.easychair.org/publications/paper/141241
10. Cauderlier, R., Dubois, C.: ML pattern-matching, recursion, and rewriting: from FoCaLiZe to Dedukti. In: Sampaio, A., Wang, F. (eds.) Theoretical Aspects of Computing - ICTAC 2016 - 13th International Colloquium, Taipei, Taiwan, ROC, October 24-31, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9965, pp. 459–468 (2016). https://doi.org/10.1007/978-3-319-46750-4_26
11. Cauderlier, R., Dubois, C.: FoCaLiZe and Dedukti to the Rescue for Proof Interoperability. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10499, pp. 131–147. Springer (2017). https://doi.org/10.1007/978-3-319-66107-0_9
12. Cauderlier, R., Halmagrand, P.: Checking Zenon Modulo Proofs in Dedukti. In: Kaliszyk, Cezary and Paskevich, Andrei (ed.) Proceedings 4th Workshop on Proof eXchange for Theorem Proving, Berlin, Germany, August 2-3, 2015. Electronic Proceedings in Theoretical Computer Science, vol. 186, pp. 57–73. Open Publishing Association, Berlin, Germany (August 2015). https://doi.org/10.4204/EPTCS.186.7
13. Cauderlier, R., Thiré, F.: Meta Dedukti. http://deducteam.gforge.inria.fr/metadedukti/
14. Dorra, A.: Équivalence Curry-Howard entre le lambda-Pi-calcul et la logique intuitionniste (2010), undergrad research intership report
15. Ebner, G., Ullrich, S., Roesch, J., Avigad, J., de Moura, L.: A metaprogramming framework for formal verification. PACMPL **1**(ICFP), 34:1–34:29 (2017). https://doi.org/10.1145/3110278
16. Gilbert, F.: Proof certificates in PVS. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10499, pp. 262–268. Springer (2017). https://doi.org/10.1007/978-3-319-66107-0_17
17. Huffman, B., Kuncar, O.: Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In: Gonthier, G., Norrish, M. (eds.) Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings. Lecture Notes in Computer Science, vol. 8307, pp. 131–146. Springer (2013). https://doi.org/10.1007/978-3-319-03545-1_9
18. Hurd, J.: First-order proof tactics in higher-order logic theorem provers. In: Archer, M., Vito, B.D., Muñoz, C. (eds.) Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003). pp. 56–68. No. NASA/CP-2003-212448 in NASA Technical Reports (Sep 2003), http://www.gilith.com/papers
19. Kaiser, J.O., Ziliani, B., Krebbers, R., Régis-Gianas, Y., Dreyer, D.: Mtac2: Typed tactics for backward reasoning in coq (2018), submitted for publication
20. McCune, W.: Prover9 and mace4 (2005–2010), http://www.cs.unm.edu/~mccune/prover9/

21. Miller, D.: A logic programming language with lambda-abstraction, function variables, and simple unification. J. Log. Comput. **1**(4), 497–536 (1991). https://doi.org/10.1093/logcom/1.4.497
22. Robinson, J.A.: A Machine-Oriented Logic Based on the Resolution Principle. J. ACM **12**(1), 23–41 (Jan 1965). https://doi.org/10.1145/321250.321253
23. Saillard, R.: Type Checking in the Lambda-Pi-Calculus Modulo: Theory and Practice. Ph.D. thesis, MINES Paritech (2015), https://pastel.archives-ouvertes.fr/tel-01299180
24. Sutcliffe, G.: The TPTP problem library and associated infrastructure. Journal of Automated Reasoning **43**(4), 337–362 (2009). https://doi.org/10.1007/s10817-009-9143-8
25. Ziliani, B., Dreyer, D., Krishnaswami, N.R., Nanevski, A., Vafeiadis, V.: Mtac: A monad for typed tactic programming in Coq. J. Funct. Program. **25** (2015). https://doi.org/10.1017/S0956796815000118
26. Zimmermann, T., Herbelin, H.: Automatic and Transparent Transfer of Theorems along Isomorphisms in the Coq Proof Assistant. CoRR **abs/1505.05028** (2015), http://arxiv.org/abs/1505.05028