ELSEVIER

Artificial Intelligence 80 (1996) 197-241

Artificial Intelligence

A metatheory of a mechanized object theory

Fausto Giunchiglia [a,b,*], Paolo Traverso [a,1]

[a] *Mechanized Reasoning Group, IRST - Istituto per la Ricerca Scientifica e Tecnologica, 38050 Povo, Trento, Italy*

[b] *Mechanized Reasoning Group, DISA, University of Trento, Via Imana 5, Trento, Italy*

Received November 1992; revised July 1994

Abstract

In this paper we propose a metatheory, MT, which represents the computation which implements its object theory, OT, and, in particular, the computation which implements deduction in OT. To emphasize this fact we say that MT is a *metatheory of a mechanized object theory*. MT has some "unusual" properties, e.g. it explicitly represents failure in the application of inference rules, and the fact that large amounts of the code implementing OT are partial, i.e. they work only for a limited class of inputs. These properties allow us to use MT to express and prove tactics, i.e. expressions which specify how to compose possibly failing applications of inference rules, to interpret them procedurally to assert theorems in OT, to compile them into the system implementation code, and, finally, to generate MT automatically from the system code. The definition of MT is part of a larger project which aims at the implementation of self-reflective systems, i.e. systems which are able to introspect their own code, to reason about it and, possibly, to extend or modify it.

1. A metatheory of a mechanized object theory

Since the seminal work by Goedel [28], metareasoning has been one of the most studied research topics in formal reasoning. Work has been done in mathematical logic (e.g. [15,39,50]), in philosophical logic (e.g. [41]), in logic programming (e.g. [5]), in many subfields of AI, such as mathematical reasoning (e.g. [11,54]), planning (e.g. [48]), programming languages (e.g. [47]) and so on. These citations are by no means exhaustive. Our interests are in theorem proving with metatheories. Similar to previous work in automated deduction, we have mechanized an object theory OT

* Corresponding author. E-mail: fausto@irst.itc.it.

[1] E-mail: leaf@irst.itc.it.

SSDI 0004-3702(95)00002-X

and its metatheory MT. The mechanization has been performed inside an interactive theorem prover called GETFOL [17]. Unlike previous work, we have defined MT to be a metatheory that takes into account the fact that OT is mechanized. To emphasize this fact, we say that MT is a *metatheory of a mechanized object theory*. This requirement can be intuitively described as follows:

- MT represents the computation which implements OT.

The words "computation" and "represent" can be formally defined, even if this is not done in this paper (but see discussion in Section 9.6). In particular, GETFOL is developed in a LISP-like programming language called HGKM [19,49,56]. Roughly speaking, by representability we mean that for any computation which can be performed by a (recursive) function in the code implementing OT, there is a deduction in MT of a corresponding "representing" formula, and vice versa. Thus, for instance, CONJ is the HGKM function in the implementation of OT such that (CONJ A) $\rightsquigarrow$ TRUE or (CONJ A) $\rightsquigarrow$ FALSE depending on whether the formula A, recorded by the data structure A, is a conjunction, where $\rightsquigarrow$ is the symbol for computation in HGKM. Then CONJ is represented in MT by the predicate symbol *Conj* such that $\vdash_{MT}$ *Conj*("A"), where "A" is the name of A if A is a conjunction and $\vdash_{MT} \neg$*Conj*("A") if this is false.

MT has also been defined to be a metatheory of provability, i.e. to be about what is provable or not provable in OT. In this perspective the above requirement becomes:

- MT represents the computation which implements deduction in OT.

Thus, for instance, fandi is the HGKM function which implements in OT the inference rule performing conjunction introduction (as described in Section 5, GETFOL implements a sequent version of natural deduction [46]). We have (fandi (GAMMA1, A) (GAMMA2, B)) $\rightsquigarrow$ (GAMMA1 GAMMA2, A AND B), where (GAMMA1, A), (GAMMA2, B), (GAMMA1 GAMMA2, A AND B) stand for the data structures recording the theorems $\Gamma_1 \longrightarrow A$, $\Gamma_2 \longrightarrow B$, $\Gamma_1, \Gamma_2 \longrightarrow A \wedge B$ of OT. Then fandi is represented in MT by a function symbol *fandi* such that $\vdash_{MT}$ *fandi*("$\Gamma_1 \longrightarrow A$", "$\Gamma_2 \longrightarrow B$") = "$\Gamma_1, \Gamma_2 \longrightarrow A \wedge B$", where "$\Gamma_1 \longrightarrow A$", "$\Gamma_2 \longrightarrow B$", "$\Gamma_1, \Gamma_2 \longrightarrow A \wedge B$" are the names in MT of the above theorems of OT. We have the further requirement that "$\Gamma_1, \Gamma_2 \longrightarrow A \wedge B$" is the unique constant for which the above equality holds. So far, we have considered successful rule applications. However, inference rules are partial functions which can be applied only if certain preconditions are satisfied, e.g. it is impossible to apply a conjunction elimination to a disjunction. This fact is left implicit in the definition on paper of a logic, but it is always implemented inside the code of theorem provers. It prevents them from asserting non-theorems. In the implementation of GETFOL, we have solved this problem by using a data structure for failure, fail and by defining new HGKM functions, called *primitive tactics*, which return the conclusion of the rules when these can be applied and fail when these cannot be applied. Primitive tactics implement the total version of inference rules by returning an explicit failure. For instance, fandeltac returns the value returned by fandel (which implements left conjunction elimination) when fandel is defined, and fail otherwise. Dually, in order to satisfy the requirement of representability, MT has a constant *fail* and new function sym-

bols, e.g. *fandeltac*, with $\vdash_{MT}$ *fandeltac*("$\Gamma \longrightarrow A \wedge B$") = *fandel*("$\Gamma \longrightarrow A \wedge B$") and $\vdash_{MT}$ *fandeltac*("$\Gamma \longrightarrow A \vee B$") = *fail*.

2. Exploiting a metatheory of a mechanized object theory

Since MT is a metatheory about deduction in a mechanized object theory, it has two main features:

(1) We can construct ground wffs and terms whose structure is in one to one correspondence with the (computation) tree constructed at the object level. In particular, there are wffs, stating the provability of an object level theorem, whose structure can be put in one to one correspondence with the computation tree of the object level proof steps which prove the theorem itself.

(2) The symbols occurring in such ground wffs and terms have corresponding symbols in the underlying HGKM mechanization. In particular, this holds for function symbols representing inference rules and primitive tactics (e.g. *fandeltac* corresponds to `fandeltac`), for predicates representing preconditions to the applicability of inference rules (e.g. *Conj* corresponds to `CONJ`), and for constants denoting symbols of the language and theorems of OT (e.g. "*A*" corresponds to `A`).

The first feature makes it possible *to express and prove tactics*, where by *tactics* we mean formulas of MT which specify how to compose primitive tactics (namely, possibly failing applications of object logic inference rules). Notice that in this paper, the word "tactic" has a different meaning from that used in most of the previous literature, e.g. in [14, 31, 32, 44], where tactics are programs written in a procedural metalanguage, e.g. in ML [34]. We call these latter tactics, *program tactics*. The second feature makes it possible *to give tactics a procedural content*, i.e. to use them to assert object level theorems (possibly proofs). This can be done in two ways. Tactics can be *interpreted*, i.e. they can be given as input to an interpreter which then asserts in OT the proved theorem. Tactics can also be *compiled* into HGKM code which can then be executed to prove theorems in OT. This process of compilation is called *flattening*. Finally, the combination of the first and the second features makes it possible to define a process, called *lifting*, which can be intuitively seen as the reverse of flattening, and which allows us to generate MT (its language and axioms) starting from the code implementing OT.

It has therefore been possible to define and implement a process where the metatheory is lifted from the code, and it is used to prove theorems representing interesting tactics which are then compiled down, or possibly interpreted in the code. As a result, derived rules can be executed like the rest of the system and used to shorten subsequent proofs. Logical manipulation at the theory level corresponds to program transformation at the system level. This reasoning cycle, which can be iterated, is schematically represented in Fig. 1 (this figure was first presented in [4]). This approach provides considerable efficiency advantages. From a computational point of view, a metatheory of a mechanized object theory allows us to compose inside a unified environment the output of code writing and metatheoretic theorem proving. From an intellectual point of view, it allows us to bridge in practice, that is inside a real running system, the gap between

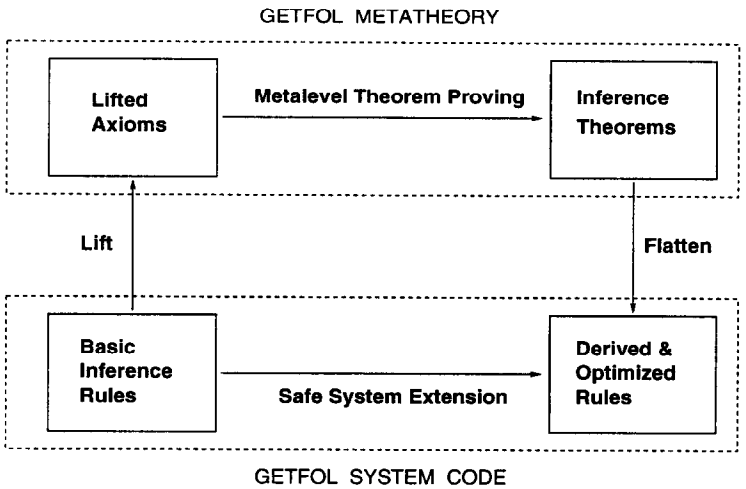

Fig. 1. The lifting-reasoning-flattening cycle.

deduction and the computation which implements deduction. This seems a first step towards "really" self-reflective systems, i.e. systems able to reason deductively about and thus, possibly, extend or modify, their underlying reasoning strategies in a provably correct way.

3. The project

This project builds on and extends Richard Weyhrauch's work on the FOL system, in particular his work on Meta, reflection principles, and simulation structures (where, using Weyhrauch's terminology, simulation structures are the mechanizable analogue of the notion of model) [54]. It can be described as an attempt to push the idea of linking computation in the code of a mechanized system and deduction in the system itself. From an implementational point of view, GETFOL has been developed on top of a reimplementation of the FOL system, described in [27]. GETFOL has, with minor variations, all the functionalities of FOL plus extensions, some described here, to allow for metatheoretic theorem proving. From a conceptual point of view, the close connection with Weyhrauch's work can be seen by analyzing the relation between OT, its mechanization, and MT, as shown in Fig. 2. MT is a metatheory of OT by construction. The code mechanizes OT by construction. The code of OT has been developed to be a finite (and partial) presentation of the model of MT, this achieves the requirement that MT represents the computation which implements OT. In fact, as described more in detail in Sections 6 and 9.1, this amounts to the following two facts. First, the interpreted constants in MT denote objects of OT which are recorded in the HGKM data structures implementing OT. Thus, for instance, "$A \wedge B$", "$\Gamma_1, \Gamma_2 \longrightarrow A \wedge B$" denote $A \wedge B$,

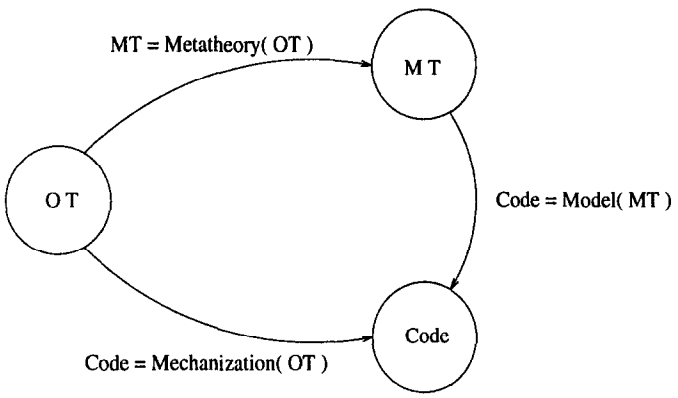

Fig. 2. OT, MT and the code.

$\Gamma_1, \Gamma_2 \longrightarrow A \wedge B$, as recorded by the data structures `(A AND B)`, `(GAMMA1 GAMMA2, A AND B)`, respectively. Second, the interpreted function and predicate symbols of MT correspond to HGKM function symbols which compute their set-theoretic meaning. In other words, HGKM functions are finite presentations of the interpretations of the corresponding application symbols in MT. Thus, for instance, the extension of *fandeltac* is computed, via the HGKM evaluator, by `fandeltac`.

However the idea of having a metatheory of a mechanized object theory is new, as are the ideas of performing lifting, flattening and of using MT for synthesizing tactics. Indeed, even though FOL had a preliminary version of the tactic interpreter (described in [54] and more in detail in [23]), FOL was never developed to the point where its code could be directly used in the metatheory, for instance to perform tactic interpretation. Most of the examples which can be found in the literature required some (often minor) ad hoc and dedicated coding. Finally, unlike Weyhrauch's metatheories (e.g. in [26, 54, 55]), we have proved the correctness of MT and of the use of the simulation structure machinery. We have in fact shown that MT, OT and its mechanization, the lifting, flattening and interpretation processes are such that tactics, when executed, will not assert non-theorems, under the hypothesis that the underlying implementation is correct (see discussion in Section 9.6).

Starting from the complete reimplementation of the FOL system described in [27], this project has been developed as two parallel subprojects strongly influencing each other. The goal of the first subproject was the mechanization of OT, i.e. the production of code which implements OT and which, at the same time, constitutes a finite presentation of the model of MT. The goal of the second project was the development of MT. These two projects have influenced each other in the sense that the mapping from computation to deduction and vice versa (and, as a consequence, e.g. the kind of tactics which can be written, how they can be executed and also the definition of the lifting and flattening functions) depends on the precise form of the axioms of MT and of the HGKM functions mechanizing OT. The problem we had to face many times was that it was impossible to map MT into the code, and vice versa. This has required multiple major recodings of GETFOL (which is more than one megabyte of code) and redefinitions of MT. The

interesting and difficult question we had to answer any time this happened was which between MT and the code had to be modified, and how. The constraint on the code is that it must do what it is supposed to do, e.g., test for theoremhood, assert a theorem, compute a conjunction, *efficiently*. It is our committment not to develop GETFOL as a toy implementation. The constraint on MT is that it must be usable *effectively*, for example, to prove theoremhood of object level theorems, synthesize tactics, reason about the structure of wffs. Currently we have succeeded in defining a metatheory MT with all the desired properties and, therefore, we have also formulated a general schema to be followed when writing the code mechanizing OT. We have recoded all of GETFOL according to this general schema. This (very slowly converging) process has lead to a situation where large parts of the code of GETFOL are really like (and look alike) axioms, and where MT contains only the necessary facts needed to prove and execute tactics.

4. Contents of the paper

The goal of this paper is to describe MT, the sense that it is a metatheory of a mechanized object theory, and the extent to which it achieves the goals described above. The details of the implementation of GETFOL are discussed only for what is needed to understand MT. The paper is structured as follows. In Section 5 we describe OT and its mechanization. This material is then used in Section 6 to describe MT, its mechanization and its connection with the code mechanizing OT. In Section 7 we show how MT can express and prove tactics. In Section 8 we show that tactics can be given a procedural interpretation by either interpreting or compiling them into the code of GETFOL. In Section 9 we give some technical results, i.e. some theorems that guarantee the correctness of our approach. The tactics considered in this paper are very simple. In Section 10 we discuss further work aimed at giving MT the desired expressibility and reasoning capabilities. In Section 11 we discuss the related work.

5. OT and its mechanization

The object theory OT is a triple OT = $\langle \mathcal{L}, \mathcal{A}x, \mathcal{R} \rangle$, where $\mathcal{L}$, $\mathcal{A}x$ and $\mathcal{R}$ are the language, the set of axioms and the set of inference rules of OT, respectively. OT is a first order classical natural deduction (ND) calculus. We consider the inference rules for $\wedge$, $\supset$, $\forall$ and $\perp$, as shown in Fig. 3 ($A, B, C, \ldots$ are well formed formulas, $\neg A$ is an abbreviation for $A \supset \perp$). The implementation of OT in GETFOL allows for a richer set of rules which contains rules for other connectives, rules for equality, derived inference rules, monadic deciders, tautology checkers, a rewriter, a semantic simplifier. Definitions and results can be easily generalized.

For various reasons, e.g. efficiency of the implementation and elegance of the proof theory, GETFOL keeps the dependencies locally to formulas. This allows one to see the GETFOL rules as rules on sequents with introduction and elimination in the post sequent. Fig. 4 describes the rules of Fig. 3 in sequent notation ($\Gamma, \Delta, \ldots$ are finite sets of formulas). (Notice that, of the structural rules, interchange and contraction are

$$\wedge I) \quad \frac{A \quad B}{A \wedge B} \qquad \wedge E) \quad \wedge E_l) \ \frac{A \wedge B}{A} \qquad \wedge E_r) \ \frac{A \wedge B}{B}$$

$$\supset I) \quad \frac{[A]}{\frac{B}{A \supset B}} \qquad \supset E) \quad \frac{A \quad A \supset B}{B}$$

$$\forall I \ x \ a) \ \frac{A}{\forall x A_x^a} \qquad \forall E \ x \ t) \ \frac{\forall x A}{A_t^x}$$

$$\perp_c) \quad \frac{\frac{[A \supset \perp]}{\perp}}{A}$$

$\forall I$ has the restriction that a must not occur in any assumption A depends on.

Fig. 3. OT inference rules presented in ND form.

$$\wedge I) \ \frac{\Gamma \longrightarrow A \quad \Delta \longrightarrow B}{\Gamma, \Delta \longrightarrow A \wedge B} \qquad \wedge E_l) \ \frac{\Gamma \longrightarrow A \wedge B}{\Gamma \longrightarrow A}$$

$$\wedge E_r) \ \frac{\Gamma \longrightarrow A \wedge B}{\Gamma \longrightarrow B}$$

$$\supset I) \ \frac{\Gamma, A \longrightarrow B}{\Gamma \longrightarrow A \supset B} \qquad \supset E) \ \frac{\Gamma \longrightarrow A \quad \Delta \longrightarrow A \supset B}{\Gamma, \Delta \longrightarrow B}$$

$$\forall I \ x \ a) \ \frac{\Gamma \longrightarrow A}{\Gamma \longrightarrow \forall x A_x^a} \qquad \forall E \ x \ t) \ \frac{\Gamma \longrightarrow \forall x A}{\Gamma \longrightarrow A_t^x}$$

$$\perp_c) \ \frac{\Gamma, A \supset \perp \ \longrightarrow \perp}{\Gamma \longrightarrow A}$$

where $\forall I$ has the restriction that a must not occur in Γ.

Fig. 4. OT inference rules presented in sequent form.

not needed as we have sets, while weakening and cut are derived inference rules.) Technically, by sequent we mean a pair (Γ, A), also written $\Gamma \longrightarrow A$, where A is the "formula of the sequent" and Γ is the set of "dependencies of the sequent". Assumptions are sequents of the form $A \longrightarrow A$. We suppose that $\supset I$ always discharges the assumption A (see Fig. 4). This can be easily generalized.

We take the notion of deduction defined in [46]. When talking about OT, we also call a deduction of a formula A depending on the possibly empty set Γ of formulae, a *proof (tree) of the sequent* $\Gamma \longrightarrow A$. We say that $\Gamma \longrightarrow A$ is a *theorem* of OT, or that $\Gamma \longrightarrow A$ is provable in OT, if and only if there exists a proof (in OT) of $\Gamma \longrightarrow A$. We write $\vdash_{\mathrm{OT}} \Gamma \longrightarrow A$ to mean that $\Gamma \longrightarrow A$ is provable in OT.

GETFOL's top level implements a listen-act-respond loop. It keeps, as part of its internal state, the proof built so far, i.e. the proved theorems and the reason why they have been proved. The proof can be inspected and manipulated by the user (via appropriate correctness preserving operations). When the action requested by the user is an inference rule application, GETFOL applies the rule, produces an error message if it fails, and it adds the proved theorem to the current proof otherwise. The details of the mechanization of GETFOL will be given in a following paper. A somewhat detailed but still incomplete description can be found in [20]. Here we mention only the relevant issues. A first issue is how to separate the code mechanizing the logic from the rest (e.g. I/O, statistics, administration facilities). A second issue is that MT must describe a subset (possibly changing for different applications) of the system functionalities and inference rules, and at the appropriate level of abstraction. Thus, for instance, it must be possible to consider only the code implementing the decision procedures without considering the code implementing natural deduction, or vice versa, or it must be possible to consider both. As a second example, when reasoning about proofs, whether wffs are implemented using pairs or lists is irrelevant. More interestingly, as Sections 6 and 7 will make clear, synthesizing tactics does not require an explicit axiomatization of what it means to be a wff. Finally, GETFOL has a lot of state (e.g. the proved theorems) which must be taken into account inside MT and, in particular, when lifting MT.

For the goals of this paper, it is sufficient to see in some detail the implementation of the inference rules. Consider for instance the GETFOL implementation of $\wedge E_l$, as reported in Figs. 5, 6 and 7. All the inference rules in GETFOL, also including the deciders, have been developed according to the general schema described in the Figs. 5, 6 and 7. This schema is idealized in the sense that it does not consider a lot of low level implementational or inessential details (e.g. the fact that the code has many failures, each recording the reason why it has been generated). However it is completely faithful to the link between MT and the HGKM code, e.g. function names, function calls, parameter passing, access to state.

Let us start with the code in Fig. 5. All this code is functional, i.e. it does not read nor access state. The code of this kind is called "computation machinery". $\wedge E_l$ is implemented by `fandel`. `fandel` never returns `fail`. `fandeltac` is the corresponding primitive tactic. Inference rules and primitive tactics are uniformly typed in the sense that they take as input the same types of objects they produce in output. Thus `fandel` takes a theorem (a `fact` in our terminology) and returns a theorem while `fandeltac` takes a theorem or a failure (a `tac` in our terminology) and returns a theorem or a failure.

Fig. 6 reports the code modifying the state of the system, i.e. `tacfproof-update` and `fproof-update`. Code of this kind is called "update machinery". `tacfproof-update` updates that part of the state which keeps the theorems and the failures generated so far, via a function call to `fproof-update` and one to `tacfproof-add-tac` (which updates the stack of theorems and failures to be processed by the current program tactic). `fproof-update` updates the current proof, via a function call to `fproof-add-fact` (which updates the current proof) or updates the standard output, via a function call to `print-error-message` (which prints an error message in the user defined standard output).

```
(DEFLAM fandeltac (tac)
 (IF (NOT (FAIL tac))
  (fandel! tac)
  fail))

(DEFLAM fandel! (fact)
 (IF (CONJ fact)
  (fandel fact)
  fail)

(DEFLAM fandel (fact)
 (fact-mak (lfand (fact-get-wff fact))
           (fact-get-deplist fact)))
```

Fig. 5. Computation machinery for left conjunction elimination.

```
(DEFLAM tacfproof-update (tac)
 (SEQ
  (tacfproof-add-tac tac)
  (fproof-update tac)))

(DEFLAM fproof-update (tac)
 (IF (FAIL tac)
  (print-error-message tac)
  (fproof-add-fact tac)))
```

Fig. 6. Update machinery for proof state and I/O state.

The code implementing the computation and the update machinery is called by the code implementing the user interface (see Fig. 7). Code of this kind is called "top level machinery". FANDEL# is called when the following command is typed to the GETFOL prompt.

```
GETFOL:: FANDEL <fact_name>;
```

GETFOL:: is printed by the system. The top level, once parsed FANDEL, understands that the user wants to apply $\wedge E_l$ and activates the routine implementing it, i.e. FANDEL#. FACT# parses a fact from the standard input, or it aborts if this is not possible. FANDEL# calls `fandel!` (and not `fandeltac`) as FACT# never returns a failure. Dually, FANDEL is called by the program tactic interpreter; TAC extracts the "current" object (a failure or a theorem) from the data structure recording the objects which must be processed by the program tactic being executed, or it aborts if this is not possible.

```
(DEFLAM FANDEL ()
  (tacfproof-update (fandeltac (TAC))) )

(DEFLAM FANDEL# ()
  (fproof-update (fandel! (FACT#))) )
```

Fig. 7. Top level user controlled machinery for left conjunction elimination.

6. MT and its mechanization

MT and OT are two distinct theories. Their implementation within GETFOL exploits the GETFOL's multitheory facilities [17]. MT is a triple MT = $\langle \mathcal{ML}, \mathcal{MA}x, \mathcal{MR} \rangle$ where $\mathcal{ML}$, $\mathcal{MA}x$ and $\mathcal{MR}$ are the language, the set of axioms and the set of inference rules of MT, respectively. MT is a first order classical ND calculus. If α is a formula of MT, then $\vdash_{\mathrm{MT}} \alpha$ means that α is provable in MT (is a theorem of MT). In the following of this section we describe $\mathcal{ML}$, $\mathcal{MA}x$, $\mathcal{MR}$. In order to keep the paper shorter, we consider the axiomatization of only two inference rules of OT, namely $\wedge E_l$ and $\forall I$. A complete definition of $\mathcal{ML}$ and $\mathcal{MA}x$ is given in Appendix A.

6.1. The language $\mathcal{ML}$

Let us start with the individual constants. In MT it must be possible to refer to certain objects of OT. We do this by adding to $\mathcal{ML}$ a new individual constant for any object of OT. These constants are the "quotation mark names" of the objects of OT [24] and are written by surrounding the string representing an object with double quotes. Thus, for instance, the quotation mark name of the individual constant c is "c", that of the individual variable x is "x", that of the formula $\forall xA$ is "$\forall xA$", that of the sequent $\Gamma \longrightarrow A$ is "$\Gamma \longrightarrow A$". In this paper we use names only for sequents, terms, variables, individual parameters and constants of OT (from now on, called generically "objects (of OT)"), which we write as s, t, x, a and c, respectively. In $\mathcal{ML}$, we have also a constant *fail* which denotes failure, as recorded by the data structure `fail` (see Fig. 5). *fail* is not the quotation mark name of an object of OT. However, analogously to all the other constants of MT, it corresponds to a data structure manipulated by the code mechanizing OT. For each inference rule and corresponding primitive tactic of OT we have an appropriate function symbol in MT:

$\wedge E_l$: *fandel*,*fandeltac* of arity 1,

$\forall I$: *falli*,*fallitac* of arity 3.

Analogously to what happens for all the other inference rules, *fandel* and *fandeltac* correspond to the HGKM functions `fandel` and `fandeltac` (see Fig. 5). $\mathcal{ML}$ has a predicate = for equality, a unary predicate *Sec* which holds of sequents, a unary predicate T for theoremhood, a unary predicate *Fail* which holds of *fail*, a predicate *Tac* which holds of *fail* and the theorems of OT, and

$\wedge E_I$: *Conj* of arity 1,

$\forall I$: *Par* of arity 1,

$\forall I$: *Var* of arity 1,

$\forall I$: *NoFree* of arity 2.

As for all the other preconditions reified in the HGKM code, *Conj* corresponds to the HGKM boolean function CONJ; *Fail* corresponds to FAIL (see Fig. 5). Finally, $\mathcal{ML}$ contains the sentential constants $\top$ (which is also an axiom) and $\bot$ for truth and falsity, respectively. $\mathcal{ML}$ has also individual variables and parameters written $x, x_1, x_2 \ldots$ and $a, a_1, a_2 \ldots$ respectively. The context always makes clear whether we are talking of variables and parameters of OT or of MT. Finally, $\mathcal{ML}$ contains the conditional term constructors **if** A **then** t_1 **else** t_2, where A is a wff and t_1, t_2 are terms. (We write **if then else** in boldface to increase the readability of formulas.)

Within GETFOL, the correspondence between the HGKM data structures and functions and the (individual, functional, predicative) constants of MT is constructed by using the simulation structure machinery and, in particular, the commands ATTACH and MATTACH (described in [17,54]). As described in detail in [54] and hinted in Section 3, the simulation structure machinery allows the user to define and use, within the system, a finite presentation of a model of the theory under consideration. ATTACH and MATTACH, in particular, implement the (mechanizable analogue) of the model interpretation function g [13]. MATTACH takes a pointer (abstractly defined) to the data structure m recording the constant m of MT and a pointer (abstractly defined) to the data structure o recording the object o of OT and stores the pair $\langle \mathtt{m}, \mathtt{o} \rangle$ as part of the state of the system. The idea is that the pair $\langle \mathtt{m}, \mathtt{o} \rangle$ records the fact that $g(m) = o$. Thus, if m is the individual constant "o", then the pair $\langle$"o", o$\rangle$ records the fact that "o" is the (quotation) mark name of o. Analgously, if m is the applicational (functional or predicative) symbol fm, then the pair $\langle \mathtt{fm}, \mathtt{fo} \rangle$, where fo is an HGKM function symbol, records the fact that the extensional (set-theoretic) characterization of fo is the interpretation of fm. Following Weyhrauch's terminology, we call the pair $\langle \mathtt{m}, \mathtt{o} \rangle$ an *attachment pair*, or simply an attachment, and we say that m is attached to o.

6.2. *The axioms $\mathcal{MA}x$*

The axioms of MT have been devised to be lifted from the mechanization of OT. However the mechanization is based on two implicit assumptions which make everything work correctly, namely that no sequent is equal to fail and that all theorems are sequents.

$$\forall x \neg(Sec(x) \wedge Fail(x)), \tag{1}$$

$$\forall x \ (T(x) \supset Sec(x)). \tag{2}$$

FAIL is implemented as an HGKM boolean function which returns TRUE when its argument computes fail and FALSE otherwise. We can therefore lift the following definition of *Fail*.

$$\forall x\ (Fail(x) \leftrightarrow x = fail). \tag{3}$$

Consider the computation machinery (Fig. 5). `fandel` is a partial function and it is called by `fandeltac` only with sequents whose wff is a conjunction. In practice, `fandel` is implemented (roughly speaking) as an HGKM `CAR` (which behaves the same as the LISP `CAR`). Thus, if it is applied to sequents whose wff is not a conjunction, `fandel` may return a wrong value (e.g. with $A \vee B$, the left disjunct A) or it may even abort (e.g. any time its argument is an HGKM atom). For all the inputs where `fandel` is not defined, `fandeltac` returns `fail`. In order to guarantee the correctness of the implementation, `fandel` never returns `fail`. This fact is captured by the following axioms.

$$\forall x\ \neg\, fandel(x) = fail, \tag{4}$$

$$\forall x_1 \forall x_2 \forall x_3\ \neg\, falli(x_1, x_2, x_3) = fail. \tag{5}$$

All the computation machinery (which, as pointed out in Section 5, is functional) is basically lifted to MT with a one to one mapping. This generates the following axioms.

$$\begin{aligned} &\forall x\ (Tac(x_1) \supset \\ &\quad fandeltac(x_1) = \textbf{if}\ \neg Fail(x_1) \wedge Conj(x_1) \\ &\qquad\qquad\qquad \textbf{then}\ fandel(x_1) \\ &\qquad\qquad\qquad \textbf{else}\ fail\ ) \end{aligned} \tag{6}$$

$$\begin{aligned} &\forall x_1 \forall x_2 \forall x_3\ (Tac(x_1) \supset \\ &\qquad fallitac(x_1, x_2, x_3) \\ &\qquad\quad = \textbf{if}\ \neg Fail(x_1) \wedge Var(x_2) \wedge Par(x_3) \wedge NoFree(x_3, x_1) \\ &\qquad\qquad\quad \textbf{then}\ falli(x_1, x_2, x_3) \\ &\qquad\qquad\quad \textbf{else}\ fail\ ) \end{aligned} \tag{7}$$

Notice that `fandeltac` is defined in terms of the function `fandel!`, which, on the other hand, has no corresponding symbol *fandel!* in MT. Indeed, in the lifting, `fandel!` is unfolded into its definiendum. An explicit definition of *fandel!* is useless for our goals as `fandel!` takes a `fact` and returns a `tac` and cannot be uniformly composed with other functions which take and produce objects of the same kind. (It is not hard to think of possible applications of *fandel!*, however, this is not one of our current goals.) Notice also that `fandeltac` is applied to objects returned by `TAC`. This generates the hypothesis $Tac(x_1)$ in axioms (6) and (7).

Consider the update machinery (Fig. 6). `tacfproof-update` updates that part of the state of `GETFOL` which stores the theorems and the failures generated so far. Dually, `TAC` (whose definition has not been given because it is not relevant) extracts objects from the system state updated by `tacfproof-update`. This part of the state of `GETFOL` approximates (in the sense that it contains a subset of) the (non-recursive) set of objects represented in MT by *Tac* (namely the set of theorems of OT union failure, see Section 9.1). This causes the lifting of `tacfproof-update` and `TAC` to *Tac*. This form of lifting can be done in general. For instance it applies also to `fproof-add-fact` and `FACT#`, which are both lifted to *T*. In fact `fproof-add-fact` adds its argument to a global variable which stores the current proof, i.e. the theorems proved so far. This global

variable approximates the (non-recursive) set of theorems of OT, represented in MT by T (see Section 9.1). The intuition is that the operations that read and update some part of the system state which approximates a given set must be lifted to the symbol representing such set. However this does not mean that `fproof-add-fact` or `FACT#` are (the finite presentation of) the interpretation of T or that `tacfproof-update` or `TAC` are (the finite presentation of) the interpretation of *Tac*. In fact none of them is attached to T or *Tac* (see Section 6.1). The fact that, for instance, `FACT#` aborts for some object, does not mean that the object is not a theorem. It might simply be a theorem which has not yet been proved. Within `fproof-update` (and also `tacfproof-update`) we have two possibilities. Either `print-error-message` is called, i.e. we have a failure, or `fproof-add-fact` is called. This generates the following axiom.

$$\forall x\ (Tac(x) \leftrightarrow T(x) \vee Fail(x)). \tag{8}$$

Consider the top level machinery (Fig. 7). Once understood how to lift the update machinery, the lifting of the top level machinery becomes a one to one mapping. Thus `FANDEL` and `FALLI` (which is defined analogously to `FANDEL`) are lifted to the following axioms. (`FANDEL#`, which could be lifted very much in the same way, has not been lifted for the same reasons as `fandel!`'s.)

$$\forall x\ (Tac(x) \supset Tac(fandeltac(x))), \tag{9}$$

$$\forall x_1 \forall x_2 \forall x_3\ (Tac(x_1) \supset Tac(fallitac(x_1, x_2, x_3))) \tag{10}$$

Finally, we have to describe the base case for deductions. Let A and B be wffs of OT.

$$T(\text{“}A \longrightarrow A\text{”}), \tag{11}$$

$$T(\text{“} \longrightarrow A\text{”}), \quad \text{if } \longrightarrow A \in \mathcal{A}x. \tag{12}$$

The above axioms describe deduction in OT and, as discussed in Section 7, allow us to express and synthesize tactics. However MT must also have axioms describing the syntax of OT, and the syntactic manipulation performed by the basic inference rules. In principle, we could generate such axioms following the same methodology used to lift the axioms above. This would allow us to use and prove universal statements about the syntax of OT, e.g. about what it means to be a well formed formula of OT. However the main goal of MT is to reason about tactics and, in particular for what concerns the syntax of OT, to be able to discriminate between when a tactic, applied to some arguments, fails or succeeds. In this perspective, as Section 9.3 will show, it is sufficient to have the ground version of such axioms and of all their consequences. Let a and x be any individual parameter and variable of OT, A and B be wffs of OT, Γ be a finite set of formulas of OT, c be any individual constant of OT and ξ be any object of OT. Then the following are axioms of MT.

$$fandel(\text{“}\Gamma \longrightarrow A \wedge B\text{”}) = \text{“}\Gamma \longrightarrow A\text{”} \tag{13}$$

$$falli(\text{“}\Gamma \longrightarrow A\text{”}, \text{“}x\text{”}, \text{“}a\text{”}) = \text{“}\Gamma \longrightarrow \forall x A_x^a\text{”}, \quad \text{where } a \text{ does not occur in } \Gamma \tag{14}$$

$Par(\text{“}a\text{”})$ (15)

$\neg Par(c)$, if c is *fail* or “ξ” and ξ is not an individual parameter of OT (16)

$Var(\text{“}x\text{”})$ (17)

$\neg Var(c)$, if c is *fail* or “ξ” and ξ is not an individual variable of OT (18)

$\neg\, c_1 = c_2$, if c_1 and c_2 are distinct individual constants (19)

$Conj(\text{“}\Gamma \longrightarrow A \wedge B\text{”})$ (20)

$\neg Conj(\text{“}\Gamma \longrightarrow A\text{”})$, if A is not a conjunction (21)

$NoFree(\text{“}a\text{”}, \text{“}\Gamma \longrightarrow A\text{”})$, if a does not appear in Γ (22)

$\neg NoFree(\text{“}a\text{”}, \text{“}\Gamma \longrightarrow A\text{”})$, if a appears in Γ (23)

$Sec(\text{“}\Gamma \longrightarrow A\text{”})$ (24)

$\neg Sec(\text{“}\xi\text{”})$, if ξ is not a sequent of OT (25)

(11)–(25) represent an infinite set of ground axioms. This is not a problem as in the actual mechanization of MT such axioms are really never asserted. For what concerns axioms (13)–(25) (and all their (ground) consequences e.g. $Conj(fandel(\text{“} \longrightarrow (A \wedge B) \wedge C\text{”})))$, the idea is to use the simulation structure machinery and to exploit the fact that the code implementing OT is a finite presentation of the model of MT. This is achieved through the command SIMPLIFY. SIMPLIFY takes in input a term or a wff and, as a first step, it computes the value denoted in the defined model (as constructed by the set of attachments, see Section 6.1). In the case of a term, the denoted value is an element of the domain, therefore SIMPLIFY asserts in MT the equality between its quotation mark name and the input term itself. In the case of a wff, the denoted value is FALSE or TRUE; in the first case SIMPLIFY asserts in MT the input wff itself, in the second, its negation. Axioms (11) and (12) cannot be asserted using SIMPLIFY, since, as seen in Section 6.1, T is not attached to any HGKM function (the reason being that T should have to be attached to a theorem prover complete for first order logic. However this would make us lose the termination of SIMPLIFY, and we want to avoid this). They are asserted by exploiting the fact that for any theorem s of OT (and, as a particular case, for any axiom and assumption) we have $\vdash_{MT} T(\text{“}s\text{”})$. This is explained in detail in Section 9.3.

Notice that an object level sequent (e.g. $\longrightarrow A$) is denoted in MT by its quotation mark name (e.g. “$\longrightarrow A$”) and by other terms that can be proved equal to its quotation mark name (e.g. $fandel(\text{“} \longrightarrow A \wedge B\text{”})$). We call the set of complex terms that are provably equal to the quotation mark name of an object of OT, the *structural descriptive names* of such object. In particular, an object level theorem can be given a structural descriptive name for each proof proving it. For instance, $fandel(\text{“} \longrightarrow A \wedge B\text{”})$ expresses the fact that $\longrightarrow A$ can be obtained by applying $\wedge E_l$ to $\longrightarrow A \wedge B$. Another structural descriptive name of $\longrightarrow A$ is $fandel(fandi(\text{“} \longrightarrow A\text{”}, \text{“} \longrightarrow B\text{”}))$ where *fandi* builds the conjunction

of two sequents. Notice moreover that writing quotation mark names using quotation marks is an option. GETFOL has no hardwired naming machinery. Using MATTACH and SIMPLIFY it is possible to give any object in OT the desired quotation mark and structural descriptive names (all the structural descriptive names being provably equal to their corresponding quotation mark name). Finally, notice that naming in GETFOL is done very efficiently, amounting to a simple table lookup in the case of quotation mark names and taking linear time in their length in the case of structural descriptive names.

To conclude, it is important to notice that MT does not represent all the code of OT but only the small part that it reasons about. In particular it is possible to lift only a subset of the inference rules implemented in GETFOL. This feature is quite useful as it allows one to use the code as a storage of axioms and to extract them selectively.

Let us consider now some examples of use of the simulation structure machinery. In all the examples of this paper we take Γ as a shorthand for $\forall x(A(x) \wedge B(x))$; s as a shorthand for the assumption $\forall x(A(x) \wedge B(x)) \rightarrow \forall x(A(x) \wedge B(x))$; Δ as a shorthand of $\forall x(A(x) \supset B(x))$; s' as a shorthand of $\forall x(A(x) \supset B(x)) \rightarrow \forall x(A(x) \supset B(x))$.

Example 6.1. Let us consider the following term:

$$\mathit{falli}(\mathit{fandel}(\mathit{falle}(\text{“}s\text{”}, \text{“}x\text{”}, \text{“}a\text{”})), \text{“}x\text{”}, \text{“}a\text{”}). \tag{26}$$

From axioms (13) and (14) (see also axiom about *falle* in Appendix A) we can prove

$$\mathit{falli}(\mathit{fandel}(\mathit{falle}(\text{“}s\text{”}, \text{“}x\text{”}, \text{“}a\text{”})), \text{“}x\text{”}, \text{“}a\text{”}) = \text{“}\Gamma \rightarrow \forall x A(x)\text{”} \tag{27}$$

by theorem proving in MT. The same result can be achieved with SIMPLIFY. The command SIMPLIFY, when executed over (26), runs a routine, called simplify, which returns the interpretation of the input expression (e.g. a term, a wff) in the defined model.

simplify(*falli*(*fandel*(*falle*(“$\Gamma \rightarrow \forall x(A(x) \wedge B(x))$”, “$x$”, “$a$”)), “$x$”, “$a$”))
= (simplify(*falli*) simplify(*fandel*(*falle*(...)))
simplify(“x”) simplify(“a”))
= (simplify(*falli*) (simplify(*fandel*) simplify(*falle*(...)))
simplify(“x”) simplify(“a”))
= (simplify(*falli*) (simplify(*fandel*) (
simplify(*falle*) simplify(“$\Gamma \rightarrow \forall x(A(x) \wedge B(x))$”)
simplify(“x”) simplify(“a”)))simplify(“x”) simplify(“a”))
= (simplify(*falli*) (simplify(*fandel*) (
falle $\Gamma \rightarrow \forall x(A(x) \wedge B(x))$ x a)) simplify(“x”) simplify(“a”))
= (simplify(*falli*) (simplify(*fandel*) $\Gamma \rightarrow A(a) \wedge B(a)$)
simplify(“x”) simplify(“a”))
= (simplify(*falli*) (fandel $\Gamma \rightarrow A(a) \wedge B(a)$)
simplify(“x”) simplify(“a”))
= (simplify(*falli*) $\Gamma \rightarrow A(a)$ simplify(“x”) simplify(“a”))
= (falli $\Gamma \rightarrow A(a)$ x a)
= $\Gamma \rightarrow \forall x A(x)$

SIMPLIFY computes the quotation mark name of the expression computed by simplify, in this case "$\Gamma \longrightarrow \forall x A(x)$", and asserts its equality with the input term as a theorem of MT, in this case (27).

Example 6.2. The functions occurring in (26) are partial. As a consequence, SIMPLIFY can be applied only to arguments where these functions are defined. To take into account failure, SIMPLIFY must be executed over terms containing primitive tactics.

$$\begin{aligned}&\texttt{simplify}(\mathit{fallitac}(\mathit{fandeltac}(\mathit{falletac}(\text{“}s'\text{”},\text{“}x\text{”},\text{“}a\text{”})),\text{“}x\text{”},\text{“}a\text{”}))\\&\quad=\texttt{simplify}(\mathit{fallitac}(\mathit{fandeltac}(\text{“}\Delta \longrightarrow A(a) \supset B(a)\text{”}),\text{“}x\text{”},\text{“}a\text{”}))\\&\quad=\texttt{simplify}(\mathit{fallitac}(\mathit{fail},\text{“}x\text{”},\text{“}a\text{”}))\\&\quad=\texttt{fail}\end{aligned}$$

Therefore, SIMPLIFY, asserts the following theorem in MT:

$$\mathit{fallitac}(\mathit{fandeltac}(\mathit{falletac}(\text{“}s'\text{”},\text{“}x\text{”},\text{“}a\text{”})),\text{“}x\text{”},\text{“}a\text{”}) = \mathit{fail}$$

Example 6.3. In MT we can prove the following wff.

$$\neg \mathit{Conj}(\mathit{falle}(\text{“}s'\text{”},\text{“}x\text{”},\text{“}a\text{”})) \tag{28}$$

We execute SIMPLIFY over $\mathit{Conj}(\mathit{falle}(\text{“}s'\text{”},\text{“}x\text{”},\text{“}a\text{”}))$.

$$\begin{aligned}&\texttt{simplify}(\mathit{Conj}(\mathit{falle}(\text{“}s'\text{”},\text{“}x\text{”},\text{“}a\text{”})))\\&\quad=\texttt{simplify}(\mathit{Conj}(\text{“}\Delta \longrightarrow A(a) \supset B(a)\text{”}))\\&\quad=\texttt{FALSE}\end{aligned}$$

Since simplify returns FALSE, then SIMPLIFY asserts the negation of the input wff, in this case (28).

6.3. The rules $\mathcal{MR}$

The set of rules of MT, $\mathcal{MR}$ consists of the same rules as OT, described in Fig. 3, for the language $\mathcal{ML}$ plus a sound and complete set of rules for equality and the rules for the introduction and elimination of conditional terms, reported in Fig. 8. Rule *if* E [*if* $E_\neg$] states that from $P(\textbf{if } A \textbf{ then } t_1 \textbf{ else } t_2)$ and A [$\neg A$], we can derive $P(t_1)$ [$P(t_2)$]. Rule *if I* states that, given a deduction of $P(t_1)$ from A and a deduction of $P(t_2)$ from $\neg A$, we can prove $P(\textbf{if } A \textbf{ then } t_1 \textbf{ else } t_2)$ ([A] denotes the fact that A is discharged from the set of wffs the conclusion depends on). The resulting theory is a conservative extension of MT [1].

7. Expressing and proving tactics

Program tactics are programs which generate proofs. They may involve any programming control construct (e.g. conditionals, loops, calls to defined program tactics). Moreover, complex program tactics are often constructed using tacticals (see for in-

$$\frac{\begin{array}{cc}[A] & [\neg A] \\ \vdots & \vdots \\ B(t_1) & B(t_2)\end{array}}{B(\textbf{if } A \textbf{ then } t_1 \textbf{ else } t_2)}\ \textbf{if } I \qquad \frac{A\ B(\textbf{if } A \textbf{ then } t_1 \textbf{ else } t_2)}{B(t_1)}\ \textbf{if } E \qquad \frac{\neg A\ B(\textbf{if } A \textbf{ then } t_1 \textbf{ else } t_2)}{B(t_2)}\ \textbf{if } E_{\neg}$$

Fig. 8. Conditional inference rules.

stance [14,32,43]). In this paper we focus on a limited class of program tactics, i.e. those that express a finite composition of proof steps. We show that

(1) there exist wffs of MT which can be put in isomorphic correspondence with program tactics, and that
(2) these wffs can be proved by building proofs where each proof step corresponds to a computation step of the program tactic.

We call these wffs, *tactics*.

7.1. Expressing tactics

Program tactics build trees of object level inference rule applications (called *sequent trees*) where either all the rules are applicable (the program tactic succeeds) or there is a rule which is not applicable (the program tactic fails). Sequent trees are formally defined as follows.

Definition 7.1 (*Sequent tree of s*). A sequent tree of a sequent s is defined inductively as follows:

(1) *Base.* For any sequent s, s is a sequent tree of s;
(2) *Step.* We have one case for each inference rule. Let $\Pi_1, \ldots, \Pi_n$ be sequent trees of $s_1, \ldots, s_n$ respectively. Let ρ be an n-ary inference rule. Then

$$\rho\frac{\begin{array}{ccc}\Pi_1 & \ldots & \Pi_n \\ s_1 & \ldots & s_n\end{array}}{s}$$

is a sequent tree of s, where s is the conclusion of the application of ρ if ρ is applicable, or "$\longrightarrow \bot$" otherwise.

We use *leaf* and *end sequent* of a sequent tree with the usual meaning. In item (2) of Definition 7.1, when ρ is not applicable, we add $\longrightarrow \bot$. The particular form of this sequent is irrelevant. We represent inference rule applications in the following uniform way.

$$\wedge E_l\ \frac{\Pi_1}{\dfrac{s_1}{s}} \qquad \forall I\ x\ a\ \frac{\Pi_1}{\dfrac{s_1}{s}}$$

We call Π the sequent tree of s built by applying an inference rule to the end sequent s_1 of Π_1. We associate to every OT sequent tree Π (Π_1) a *tactic term* τ_π (τ_{π_1}). Tactic terms are defined inductively over the structure of sequent trees.

Definition 7.2 (*Tactic term of* Π). The tactic term τ_π of the sequent tree Π of s is defined inductively over the structure of Π.

(1) *Base.*

$$\tau_\pi = \begin{cases} \text{“}s\text{”}, & \text{if } s \text{ is an assumption or an axiom,} \\ \mathit{fail}, & \text{otherwise.} \end{cases}$$

(2) *Step.*

(a) $(\wedge E_l)$: $\tau_\pi = \mathit{fandeltac}(\tau_{\pi_1})$,

(b) $(\forall I x a)$: $\tau_\pi = \mathit{fallitac}(\tau_{\pi_1}, \text{“}x\text{”}, \text{“}a\text{”})$.

Example 7.3. Let us consider the following proof tree Π (the axioms for $\forall E$ are in Appendix A):

$$\begin{array}{ll} & \dfrac{\Gamma \longrightarrow \forall x A(x) \wedge B(x)}{\Gamma \longrightarrow A(a) \wedge B(a)} \\ \forall E & \\ \wedge E_l & \dfrac{\Gamma \longrightarrow A(a)}{\Gamma \longrightarrow \forall x A(x)} \\ \forall I & \end{array}$$

τ_π is $\mathit{fallitac}(\mathit{fandeltac}(\mathit{falletac}(\text{“}s\text{”}, \text{“}x\text{”}, \text{“}a\text{”})), \text{“}x\text{”}, \text{“}a\text{”})$.

Example 7.4. Let us consider the following sequent tree Π (notice that Π is not a proof tree):

$$\begin{array}{ll} & \dfrac{\Delta \longrightarrow \forall x A(x) \supset B(x)}{\Delta \longrightarrow A(a) \supset B(a)} \\ \forall E & \\ \wedge E_l & \dfrac{\longrightarrow \bot}{\longrightarrow \forall x \bot} \\ \forall I & \end{array}$$

τ_π is $\mathit{fallitac}(\mathit{fandeltac}(\mathit{falletac}(\text{“}s'\text{”}, \text{“}x\text{”}, \text{“}a\text{”})), \text{“}x\text{”}, \text{“}a\text{”})$.

Tactic terms contain constants which denote either the leaves of the corresponding sequent tree or failure. Program tactics, however, take arguments which get instantiated at execution time. We are therefore interested in a “generalization” of tactic terms, where constants are replaced by variables, and which represent proof structures independently of the leaves of sequent trees. Technically, this is obtained as follows. We write tactic terms τ_π as $\tau_\pi[c_1, \ldots, c_n]$, where $c_1, \ldots, c_n$ are the individual constants appearing in the term. Each c_i in $c_1, \ldots, c_n$ may be the quotation mark name of a sequent, the constant *fail*, or the quotation mark name of a possible parameter of a rule application (like “x” and “a” in $\mathit{fallitac}(\text{“}s\text{”}, \text{“}x\text{”}, \text{“}a\text{”})$). Let $x_1, \ldots, x_n$ be individual variables of MT. By $\tau_\pi[x_1, \ldots, x_n]$ we mean the term obtained by replacing the constants $c_1, \ldots, c_n$ in $\tau_\pi[c_1, \ldots, c_n]$ with the variables $x_1, \ldots, x_n$, respectively. We can now define the notion of *tactic*.

Definition 7.5 (*Tactic*). Let $c_1, \ldots, c_n$ be constants and $\tau_\pi[c_1, \ldots, c_n]$ a tactic term. Let $c_1, \ldots, c_m$, with $m \leqslant n$, be all and the only constants in $c_1, \ldots, c_n$ that are either quotation mark names of sequents or *fail*. Then a tactic is any wff of the form:

$$\forall x_1 \ldots \forall x_n \ (Tac(x_1) \wedge \cdots \wedge Tac(x_m) \supset Tac(\tau_\pi[x_1, \ldots, x_n])). \tag{29}$$

Axioms (9) and (10) are tactics. In Section 6 we have explained how they are in a precise correspondence with the primitive tactics for $\wedge E_l$ and $\forall I$. The same correspondence exists in general between complex program tactics and tactics. Indeed, each function symbol in $\tau_\pi[x_1, \ldots, x_n]$ corresponds to a primitive tactic. Consider for instance the following example.

Example 7.6.

$$\forall x_1 x_2 x_3 \ (Tac(x_1) \supset Tac(fallitac(fandeltac(falletac(x_1, x_2, x_3)), x_2, x_3))). \tag{30}$$

(30) corresponds to the program tactic that composes the three primitive tactics `falletac`, `fandeltac` and `fallitac`.

7.2. Proving tactics

Tactics which correspond to primitive tactics are axioms of MT. From these axioms we can prove tactics which correspond to compositions of primitive tactics.

Example 7.7. Consider tactic (30). From the following axiom of MT (see Appendix A)

$$\forall x_1 \forall x_2 \forall x_3 \ (Tac(x_1) \supset Tac(falletac(x_1, x_2, x_3))) \tag{31}$$

and axiom (9) we can prove

$$\forall x_1 \forall x_2 \forall x_3 \ (Tac(x_1) \supset Tac(fandeltac(falletac(x_1, x_2, x_3)))). \tag{32}$$

Then, from (32) and axiom (10) we can prove (30). Notice that the proof starts from wff (31) which corresponds to `falletac`. It then introduces *fandeltac* and finally *fallitac*.

The proof in Example 7.7 suggests one possible general way to prove tactics, namely to build a proof where each step corresponds to a computation step of the corresponding program tactics. Intuitively, the proof in Example 7.7 corresponds to checking that the object constructed by the composition of `falletac`, `fandeltac` and `fallitac` is of type *Tac*, i.e. that it is either a theorem or a failure. This is similar to what happens in tactic-based theorem provers, where program tactics construct theorems only by applying primitive tactics. Nevertheless, MT can prove statements which can be executed to assert theorems and which do not contain primitive tactics. For instance,

$$\forall x \ (Tac(x) \wedge Conj(x) \supset Tac(fandel(x))) \tag{33}$$

where *fandel* corresponds to `fandel`, is a theorem of MT. Intuitively, (33) states that, under the proper conditions, i.e. when the argument is a conjunction, `fandel` can be used safely to construct a theorem. Similarly, the following are also theorems of MT.

$$\forall x\ (T(x) \wedge Conj(x) \supset T(fandel(x))) \tag{34}$$

$$\begin{aligned}\forall x_1 \forall x_2 \forall x_3\ &(T(x_1) \wedge Var(x_2) \wedge Par(x_3) \wedge NoFree(x_3, x_1) \\ &\supset T(falli(x_1, x_2, x_3)))\end{aligned} \tag{35}$$

$$\begin{aligned}\forall x_1 \forall x_2 \forall x_3\ &(T(x_1) \\ &\wedge Var(x_2) \wedge Term(x_3) \wedge Forall(x_1) \\ &\wedge Conj(falle(x_1, x_2, x_3)) \\ &\wedge Var(x_2) \wedge Par(x_3) \wedge NoFree(x_3, fandel(falle(x_1, x_2, x_3))) \\ &\supset T(falli(fandel(falle(x_1, x_2, x_3)), x_2, x_3)))\end{aligned} \tag{36}$$

The proof of (34) and (35) can be easily performed from tactics (9) and (10) by unfolding *fandeltac* and *fallitac* and factoring out the conditions of applicability of the rules. (36) can be proved in an analogous way. (34), (35) describe the object level rules $\wedge E_l$ and $\forall I$. (36) describes the derived rule that applies $\forall E$, $\wedge E_l$ and $\forall I$ in the given order. Notice that (34), (35) and (36) describe explicitly the preconditions of applicability of the corresponding (derived) rule. In (36) we have the preconditions of $\forall E$ (second line), the preconditions of $\wedge E_l$ (third line) and those of $\forall I$ (fourth line). The term $falli(fandel(falle(x_1, x_2, x_3)), x_2, x_3)))$ (fifth "line") describes how rules get composed. (34), (35) and (36) say nothing of what happens when a rule which cannot be applied is applied. They do not take into account the code dealing with failures. In this sense, they are closer to the "usual" on paper metalevel descriptions of "non-mechanized" (derived) rules.

Finally, since tactics are theorems of MT and correspond to program tactics, we can perform logical manipulation which corresponds to code transformation, thus optimizing program tactics. For instance, we can prove a wff which is logically equivalent to tactic (30) and which corresponds to a program tactic which avoids redundant tests.

Example 7.8. We can prove in MT

$$\begin{aligned}\forall x_1 \forall x_2 \forall x_3\ &(Tac(x_1)\ \supset \\ &fallitac(fandeltac(falletac(x_1, x_2, x_3)), x_2, x_3) = \\ &\quad \textbf{if}\ \neg Fail(x_1) \\ &\quad \wedge Var(x_2) \wedge Term(x_3) \wedge Forall(x_1) \\ &\quad \wedge Conj(falle(x_1, x_2, x_3)) \\ &\quad \wedge Par(x_3) \wedge NoFree(x_3, fandel(falle(x_1, x_2, x_3))) \\ &\quad\ \ \textbf{then}\ falli(fandel(falle(x_1, x_2, x_3)), x_2, x_3) \\ &\quad\ \ \textbf{else}\ fail\ )\end{aligned} \tag{37}$$

The proof is performed by rewriting first $fallitac(fandeltac(falletac(x_1, x_2, x_3)), x_2, x_3)$ according to axioms (6), (7) and the axiom about *falletac* in Appendix A, under the assumption $Tac(x_1)$. We assume the wff in the **if** condition in wff (37) and its negation.

This allows us to eliminate conditional terms by applying **if** E and **if** $E_{\neg}$. We can thus obtain

$$fallitac(fandeltac(falletac(x_1, x_2, x_3)), x_2, x_3) = falli(fandel(falle(x_1, x_2, x_3)), x_2, x_3)$$

depending on the **if** conditions in wff (37) (and $Tac(x_1)$), and

$$fallitac(fandeltac(falletac(x_1, x_2, x_3)), x_2, x_3) = fail$$

depending on the negation of the **if** conditions in wff (37) (and $Tac(x_1)$). We can thus apply **if** I discharging the assumptions and $\supset I$ discharging $Tac(x_1)$. While (30) corresponds to a program tactic which calls FAIL three times, and Var twice, (37) corresponds to a program tactic which calls these routines only once.

Example 7.8 is very simple. The underlying intuition is that MT could be used (possibly with extensions) to optimize code in more significant ways (see discussion about Isabelle-like tactics in Section 10).

8. Executing tactics

Tactics can be given a procedural content and thus used to assert object level theorems. This can be done by interpreting them or by compiling (flattening) them into program tactics.

8.1. Interpreting tactics

Consider a generic tactic

$$\forall x_1 \ldots \forall x_n \ (Tac(x_1) \wedge \cdots \wedge Tac(x_m) \supset Tac(\tau_\pi[x_1, \ldots, x_n])).$$

By "tactic interpretation" we mean a process by which a tactic is fed into an interpreter which asserts an object level theorem or fails. This process is performed in the following four steps.

Step 1. Perform a sequence of forall eliminations, to obtain

$$Tac(c_1) \wedge \cdots \wedge Tac(c_m) \supset Tac(\tau_\pi[c_1, \ldots, c_n])$$

where $c_1, \ldots, c_m, \ldots, c_n$ are constants naming objects of OT. This is the dual operation, in a procedural metalanguage, of typing the call to a program tactic on a set of arguments.

Step 2. If any c_i in $c_1, \ldots, c_m$ is either *fail* or the name of a sequent that has been proved in OT, then deduce

$$Tac(\tau_\pi[c_1, \ldots, c_n])$$

as a theorem of MT; otherwise, stop. This is the dual operation of testing that some of the arguments of the program tactic are of the right type, i.e. that they are either theorems or failure.

Step 3. Run `simplify` over the term $\tau_\pi[c_1, \ldots, c_n]$. If it returns a data structure implementing an object level sequent s, then deduce $T(\text{“}s\text{”})$ in MT. If it returns `fail`, stop. This is the dual operation of executing the body of the program tactic with instantiated arguments.

Step 4. From T(“s”), assert s as a theorem of OT. This is the dual operation of asserting the result of the execution of a program tactic.

Example 8.1. Let us consider tactic (30). In Step 1, we instantiate it by performing three forall eliminations and by replacing x_1, x_2 and x_3 with “s”, “x” and “a”, respectively. We obtain the following formula.

$$Tac(\text{“}s\text{”}) \supset Tac(fallitac(fandeltac(falletac(\text{“}s\text{”}, \text{“}x\text{”}, \text{“}a\text{”})), \text{“}x\text{”}, \text{“}a\text{”})).$$

In Step 2, we exploit the fact that $\vdash_{OT} s$:

$$Tac(fallitac(fandeltac(falletac(\text{“}s\text{”}, \text{“}x\text{”}, \text{“}a\text{”})), \text{“}x\text{”}, \text{“}a\text{”})). \tag{38}$$

The term in (38) is the tactic term of the sequent tree in Example 7.3. Step 3 runs `simplify` over $fallitac(fandeltac(falletac(\text{“}s\text{”}, \text{“}x\text{”}, \text{“}a\text{”})), \text{“}x\text{”}, \text{“}a\text{”})$, that returns $\Gamma \longrightarrow \forall x A(x)$. Then we have

$$T(\text{“}\Gamma \longrightarrow \forall x A(x)\text{”}).$$

Step 4 asserts $\Gamma \longrightarrow \forall x A(x)$ as a theorem of OT.

Example 8.2. Let us consider now an execution that fails. We take the same tactic as before, but in Step 1 we instantiate it to obtain a ground wff that contains the tactic term of Example 7.4. We can perform Step 2 as in Example 8.1 to obtain:

$$Tac(fallitac(fandeltac(falletac(\text{“}s'\text{”}, \text{“}x\text{”}, \text{“}a\text{”})), \text{“}x\text{”}, \text{“}a\text{”})).$$

Step 3 runs `simplify` over $fallitac(fandeltac(falletac(\text{“}s'\text{”}, \text{“}x\text{”}, \text{“}a\text{”})), \text{“}x\text{”}, \text{“}a\text{”})$, that returns `fail` (see Example 6.2). Then the process stops and Step 4 is not performed.

Another possibility is to interpret metalevel statements that describe (derived) rules, e.g. wffs (34), (35) and (36). Their interpretation is similar to the interpretation of tactics. The first step is the same as Step 1. In the second step, wffs of the form $T(\text{“}s\text{”})$, where s is a theorem of OT, get factored out of the ground wff. The third step applies `simplify` to each of the atomic ground wffs whose main predicate symbol is not T. If it returns `TRUE` for all of them, then `simplify` is applied to the term argument of T. Otherwise, the process stops. The last step is the same as Step 4.

Example 8.3. Consider wff (36). After the first two steps we have

$$\begin{aligned} &Var(\text{“}x\text{”}) \wedge Term(\text{“}a\text{”}) \wedge Forall(\text{“}s\text{”}) \\ &\wedge Conj(falle(\text{“}s\text{”}, \text{“}x\text{”}, \text{“}a\text{”})) \\ &\wedge Var(\text{“}x\text{”}) \wedge Par(\text{“}a\text{”}) \wedge NoFree(\text{“}x\text{”}, fandel(falle(\text{“}s\text{”}, \text{“}x\text{”}, \text{“}a\text{”}))) \\ &\quad \supset T(falli(fandel(falle(\text{“}s\text{”}, \text{“}x\text{”}, \text{“}a\text{”})), \text{“}x\text{”}, \text{“}a\text{”})) \end{aligned} \tag{39}$$

The third step applies `simplify` to the conjuncts. For each conjunct in (39), it returns `TRUE`. `simplify` is then applied to $falli(fandel(falle(\text{"}s\text{"},\text{"}x\text{"},\text{"}a\text{"})),\text{"}x\text{"},\text{"}a\text{"})$. It returns $\Gamma \longrightarrow \forall x A(x)$ (see Example 6.1), which is then asserted as a theorem of OT. If x_1 gets instantiated with s', then `simplify` returns `FALSE` when applied to $Conj(falle(\text{"}s\text{"},\text{"}x\text{"},\text{"}a\text{"}))$ (Example 6.3). Therefore the process stops.

8.2. *Flattening tactics*

By flattening we mean a process which takes (certain) theorems of MT and generates program tactics in the `HGKM` code. As seen in Section 5, the code implementing a (primitive) program tactic is composed of three parts: the computation machinery, the top level machinery and the update machinery. Flattening must generate the computation and the top level machinery. The update machinery is the same for all inference rules. In MT it is possible to "split" a tactic into two wffs, each of them corresponding to the two parts of the code that must be generated.

$$\forall x_1 \ldots \forall x_n \; newtac(x_1, \ldots, x_n) = \tau_\pi[x_1, \ldots, x_n], \tag{40}$$

$$\forall x_1 \ldots \forall x_n \; (Tac(x_1) \wedge \cdots \wedge Tac(x_m) \supset Tac(newtac(x_1, \ldots, x_n))). \tag{41}$$

(40) is the definition of a new function symbol *newtac*. We obtain (41) by rewriting $\tau_\pi[x_1, \ldots, x_n]$ in (29) with (40). We can now generate the code of a new program tactic `newtac` corresponding to *newtac*. We flatten its computation machinery from (40). We flatten its top level machinery from (41). The flattening process exploits the attachments of MT applicative symbols to `HGKM` functions. Given the attachments, the flattening of (40) and (41) consists of a simple syntactical translation. Consider the following example.

Example 8.4. From tactic (30) we can prove

$$\begin{aligned} &\forall x_1 \forall x_2 \forall x_3 \; distac(x_1, x_2, x_3) \\ &\qquad = fallitac(fandeltac(falletac(x_1, x_2, x_3)), x_2, x_3), \end{aligned} \tag{42}$$

$$\forall x_1 \forall x_2 \forall x_3 \; (Tac(x_1) \supset Tac(distac(x_1, x_2, x_3))) \tag{43}$$

where *distac* corresponds to a program tactic which implements distributivity (of the universal quantifier over conjunctions). (42) gets flattened to

```
(DEFLAM distac (x1 x2 x3)
    (fallitac (fandeltac (falletac x1 x2 x3)) x2 x3) )
```

Notice that the universal statement is translated into a function definition. The universally quantified variables x_1, x_2 and x_3 become the arguments `x1`, `x2` and `x3` of the function definition. The function symbol *distac* gets translated into the `HGKM` symbol `distac` and *distac* is attached to `distac`. A simple syntactic translation is performed on the tactic term to obtain the body of the definition of `distac`. The top level machinery for `distac` is flattened from (43) to obtain

```
(DEFLAM DIS ()
```

```
(tacfproof-update (distac (TAC) (TERM) (TERM))) )
```

The name given to the top level function DIS is arbitrary. Flattening always builds the composition of the update routine tacfproof-update and of the functional part of the program tactic (in this case distac). The arguments of distac are constructed according to the hypotheses of the implication in (43). Since we have $Tac(x_1)$, the first argument gets extracted by TAC. TERM is a routine for the extraction of terms of no specific type.

We can also flatten logical manipulations of tactics and thus generate optimized code. Consider the following example.

Example 8.5. In Section 7 we have manipulated wff (30) and obtained wff (37). From (37) and (42) we can prove

$$\begin{aligned}
&\forall x_1 \forall x_2 \forall x_3\ (Tac(x_1) \supset \\
&\qquad distac(x_1, x_2, x_3) = \\
&\qquad\quad \textbf{if } \neg Fail(x_1) \\
&\qquad\qquad \wedge Var(x_2) \wedge Par(x_3) \wedge Forall(x_1) \\
&\qquad\qquad \wedge Conj(falle(x_1, x_2, x_3)) \\
&\qquad\qquad \wedge NoFree(x_3, fandel(falle(x_1, x_2, x_3))) \\
&\qquad\qquad\quad \textbf{then } falli(fandel(falle(x_1, x_2, x_3)), x_2, x_3) \\
&\qquad\qquad\quad \textbf{else } fail)
\end{aligned} \tag{44}$$

From (44) we can flatten the computation machinery of distac (and leave unchanged the top level machinery). The result is:

```
(DEFLAM distac (x1 x2 x3)
    (if (AND (NOT (FAIL x1))
             (VAR x2) (PAR x3) (FORALL x1)
             (CONJ (falle x1 x2 x3))
             (NOFREE (fandel (falle x1 x2 x3))))
        (falli (fandel (falle x1 x2 x3)) x2 x3)
        fail))
```

The flattening process performs the same syntactic translations as in Example 8.4, namely it translates function and predicate symbols according to their attachments, it translates the conditional term **if** ... **then** ... **else** ... into the conditional construct (if ...) and the connectives $\wedge$, $\neg$ into AND, NOT, respectively. Notice that this program tactic is an optimized version of distac flattened in Example 8.4. When flattened, this definition will replace the previous definition.

Once flattened, program tactics get executed in the standard way, i.e. the user can type the call to the program tactic with proper arguments, the arguments get type checked (e.g. by TAC), the body of the program tactic gets executed, and the state of the system gets affected by the update machinery. Executing the HGKM code obtained by flattening a tactic gives the same result as interpreting the tactic in MT.

Finally, notice that program tactics can be lifted into axioms of MT by a process which is the inverse of flattening. For instance, the user can handcode `distac` and `DIS` as in Example 8.4. This code can be lifted into wffs (42) and (43). Once MT has been lifted for the first time, lifting and flattening involves only the computation machinery and the top level machinery function definitions.

9. Everything works – some technical results

In this section we give some results which guarantee the correctness of the solutions proposed in Sections 6, 7 and 8. The proofs of theorems are in Appendix B. The results presented are technically not hard to prove. As described in Section 3, the hard work has been in stating them, i.e. in defining a metatheory and a mechanization of OT such that these results could be proved. They are needed to show that the proposed framework has all the desired properties. This section is mainly technical but not completely, as the results are discussed in relation to the goals of the paper.

9.1. MT is consistent

We define an interpretation $\mathcal{M} = \langle \mathcal{D}, g \rangle$ of $\mathcal{ML}$, where $\mathcal{D}$ is the domain of interpretation and g is the interpretation function. The domain of interpretation $\mathcal{D}$ includes a set (called $\mathcal{D}_o$) of objects of OT. The domain contains two special elements E and F. E intuitively denotes the value "undefined" and is used to handle partialness. We use F to interpret failure, i.e. the constant *fail* of MT.

Definition 9.1. The interpretation $\mathcal{M}$ of $\mathcal{ML}$ is the pair $\langle \mathcal{D}, g \rangle$. $\mathcal{D} = \mathcal{D}_o \cup \{\mathrm{E}\} \cup \{\mathrm{F}\}$. $\mathcal{D}_o$ is the set of terms, wffs and sequents of OT. E and F are distinct from any other element of $\mathcal{D}$. g is defined as follows:

(1) $g(\text{“}s\text{”}) = s$, where s is any sequent of OT.
(2) $g(\text{“}w\text{”}) = w$, where w is any wff of OT.
(3) $g(\text{“}t\text{”}) = t$, where t is any term of OT.
(4) $g(\mathit{fail}) = \mathrm{F}$.
(5) $g(\mathit{Sec})$ is the set of sequents of OT.
(6) $g(=)$ is the identity relation over $\mathcal{D}$.
(7) $g(\mathit{Par})$ is the set of individual parameters of OT.
(8) $g(\mathit{Var})$ is the set of individual variables of OT.
(9) $g(\mathit{Conj})$ is the set of sequents of OT whose formula is a conjunction.
(10) $g(\mathit{NoFree})$ is the relation over $\mathcal{D}^2$ such that $(d_1, d_2) \in g(\mathit{NoFree})$ iff d_1 is an individual parameter of OT, d_2 is a sequent $\Gamma \longrightarrow A$ of OT and d_1 does not appear in Γ.
(11) $g(T) = T_{\mathrm{OT}}$, where T_{OT} is the set of theorems of OT.
(12) $g(\mathit{fandel})$ is a function from $\mathcal{D}$ to $\mathcal{D}$ such that, for any $d \in \mathcal{D}$

$$g(\mathit{fandel})(d) = \begin{cases} \Gamma \longrightarrow A, & \text{if } d \text{ is } \Gamma \longrightarrow A \wedge B, \\ \mathrm{E}, & \text{otherwise.} \end{cases}$$

(13) $g(falli)$ is a function from $\mathcal{D}^3$ to $\mathcal{D}$ such that, for any $d_1, d_2, d_3 \in \mathcal{D}$

$$g(falli)(d_1,d_2,d_3) = \begin{cases} \Gamma \longrightarrow \forall x A^a_x, & \text{if } d_1 \text{ is } \Gamma \longrightarrow A,\ d_2 \text{ is } a,\ d_3 \text{ is } x \\ & \text{and } a \text{ does not appear in } \Gamma, \\ \mathtt{E}, & \text{otherwise.} \end{cases}$$

(14) $g(fandeltac)$ is a function from $\mathcal{D}$ to $\mathcal{D}$ such that, for any $d \in \mathcal{D}$

$$g(fandeltac)(d) = \begin{cases} g(fandel)(d), & \text{if } d \in T_{OT} \text{ and } d \in g(Conj), \\ \mathtt{F}, & \text{if } d \in T_{OT} \cup \{\mathtt{F}\} \\ & \text{and } d \notin g(Conj), \\ \mathtt{E}, & \text{otherwise.} \end{cases}$$

(15) $g(fallitac)$ is a function from $\mathcal{D}^3$ to $\mathcal{D}$ such that, for any $d_1, d_2, d_3 \in \mathcal{D}$

$$g(fallitac)(d_1,d_2,d_3) = \begin{cases} g(falli)(d_1,d_2,d_3), \\ \quad \text{if } d_1 \in T_{OT},\ d_2 \in g(Var),\ d_3 \in g(Par) \\ \quad \text{and } (d_3,d_1) \in g(NoFree), \\ \mathtt{F}, \\ \quad \text{if } d_1 \in T_{OT} \cup \{F\} \text{ and } (d_2 \notin g(Var) \text{ or} \\ \quad d_3 \notin g(Par) \text{ or } (d_3,d_1) \notin g(NoFree)), \\ \mathtt{E}, \\ \quad \text{otherwise.} \end{cases}$$

Wffs and terms get interpreted according to the usual standard tarskian semantics. The semantics of conditional terms is as follows. The value of **if** A **then** t_1 **else** t_2 is the value of t_1 if A is true, and the value of t_2 otherwise.

Theorem 9.2. *$\mathcal{M}$ is a model of MT.*

Theorem 9.2 proves the consistency of MT. Moreover, since $\mathcal{M}$ is a model of MT, we have that

$$\text{If } \vdash_{MT} T(\text{“}s\text{”}) \text{ then } \vdash_{OT} s \tag{45}$$

for any sequent s of OT. We say that MT is correct with respect to OT.

It should now be clear in which sense the code of OT has been developed to be *a finite presentation* of the model of MT. Compare Definition 9.1 of $\mathcal{M}$ and the description of the mechanization of OT given in Section 5. The HGKM functions which perform deduction in OT (e.g. `fandel`, `fandeltac`, `CONJ`) are (implemented to be) finite presentations of the relations assigned by g (as defined above) to the corresponding application symbols (e.g. $g(fandel)$, $g(fandeltac)$, $g(Conj)$). We can thus use the simulation structure machinery to implement the mechanizable analogous of the interpretation in the model $\mathcal{M}$. The commands `ATTACH` and `MATTACH` implement the (mechanizable analogue) of g. `simplify` tests the truth of wffs in $\mathcal{M}$, i.e. it implements $\models_{\mathcal{M}}$. It is important to notice that the code of OT is a presentation of only a partial model of MT. In particular *Tac* and T are left uninterpreted (see discussion in Section 6).

In Definition 9.1, we have introduced E to interpret function symbols (e.g. *fandel* and *falli*) which correspond to HGKM partial functions (e.g. `fandel` and `falli`). Partialness is a general characteristic of a large amount of the code of GETFOL (and of any running system). As it can be noticed from Definition 9.1, points (14) and (15), also `fandeltac` and `fallitac` are partial functions defined only over theorems or failures (corresponding to the set $(T_{OT} \cup \{\mathrm{F}\}) \subseteq \mathcal{D}$). This is the case also for the tacticals implemented in GETFOL [25] (which are defined only over program tactics) and for all the code implementing destructors and constructors of logical syntactical categories, e.g. wffs and terms. Partialness allows us to achieve efficiency as the code does not have to test and decide for all the possible inputs.

Extending the domain with E to handle partial functions is a well known standard technique (see, for instance, [12,42]). One essential difference is that we have two distinct special elements, E and F. From a theoretical point of view, we could have constructed a model and a metatheory where E and F are collapsed into a unique element. The problem is that the mechanization of OT is not a finite presentation of this model. The distinction between E and F is important in order to define a correspondence between MT, its model $\mathcal{M}$ and the code implementing deduction in OT (as shown in Fig. 2). E is not denoted by any symbol in the language of MT and is not implemented by any data structure in the GETFOL code. It is used to capture in the model "defined on paper" the fact that some programs are partial. On the contrary, F is denoted by *fail* in MT and is implemented by the data structure `fail` in the GETFOL code. We say that `fail` is "a witness of observable failures".

9.2. *The use of SIMPLIFY is correct*

In Section 6 we use SIMPLIFY to assert axioms (13)–(25) and all their (ground) consequences. We show here that this use of the simulation structure machinery is sound. First notice that SIMPLIFY cannot be applied to all the (ground) terms or wffs of MT, since in MT we have symbols that are attached to partial functions. For instance, SIMPLIFY *fandel*("$\longrightarrow A \vee B$") would return a wrong value and SIMPLIFY *Conj*("x"), where x is a variable of OT, would abort (see discussion in Section 6). In order to guarantee soundness, we apply SIMPLIFY only to a subset S_t of terms of MT, called (the set of) *simplifiable terms*, and a subset S_w of wffs of MT, called (the set of) *simplifiable wffs*. Roughly speaking, these sets contain all and only the ground terms and atomic ground wffs which are well sorted. In the following, we write $[[t]]_{\mathcal{M}}$, to mean the element of $\mathcal{D}$ denoted by the term t of MT.

Definition 9.3 (*Simplifiable terms*).

(1) Let c be a constant of MT. Then $c \in S_t$.
(2) If $t \in S_t$ and $[[t]]_{\mathcal{M}} \in g(Conj)$, then $fandel(t) \in S_t$.
(3) If $t_1, t_2, t_3 \in S_t$, $([[t_3]]_{\mathcal{M}}, [[t_1]]_{\mathcal{M}}) \in g(NoFree)$ and $[[t_2]]_{\mathcal{M}} \in g(Var)$, then $falli(t_1, t_2, t_3) \in S_t$.
(4) If $t \in S_t$ and $[[t]]_{\mathcal{M}} \in T_{OT} \cup \{\mathrm{F}\}$, then $fandeltac(t) \in S_t$.
(5) If $t_1, t_2, t_3 \in S_t$ and $[[t_1]]_{\mathcal{M}} \in T_{OT} \cup \{\mathrm{F}\}$, then $fallitac(t_1, t_2, t_3) \in S_t$.

Definition 9.4 (*Simplifiable wffs*).

(1) If $t_1, t_2 \in S_t$, then $t_1 = t_2 \in S_w$.
(2) If $t \in S_t$, then $Sec(t) \in S_w$.
(3) If $t \in S_t$ and $[[t]]_{\mathcal{M}} \in g(Sec) \cup \{\mathbf{F}\}$, then $Fail(t) \in S_w$.
(4) If $t \in S_t$ and $[[t]]_{\mathcal{M}} \in g(Sec)$, then $Conj(t) \in S_w$.
(5) If $t \in S_t$, then $Par(t) \in S_w$.
(6) If $t \in S_t$, then $Var(t) \in S_w$.
(7) If $t_1, t_2 \in S_t$, $t_1 \in g(Par)$ and $t_2 \in g(Sec)$, then $Nofree(t_1, t_2) \in S_w$.

Notice that even if $t \in S_t$, $T(t) \notin S_w$ and $Tac(t) \notin S_w$ also in the case where t is of the correct type, i.e. it denotes theorems or theorems and failures, respectively. This is only because T and *Tac* are not attached to anything.

The soundness of the operation performed by `SIMPLIFY` is not obvious as in general the set of provable formulas is a subset of the set of true formulas. The following theorem guarantees that this is not the case.

Theorem 9.5 (Correctness of simplify). *Let* $w \in S_w$. *Then* $\models_{\mathcal{M}} w \Longrightarrow \vdash_{\mathrm{MT}} w$ *and* $\not\models_{\mathcal{M}} w \Longrightarrow \vdash_{\mathrm{MT}} \neg w$.

Notice that, from a purely theoretical point of view, with minor modifications of the ground axioms, Theorem 9.5 can be stated for all the ground atomic wffs that do not contain T and *Tac*, and not only limited to simplifiable wffs. For instance, we could extend the set of ground axioms to include $\neg Conj(\text{“}x\text{”})$ and thus have $\not\models_{\mathcal{M}} Conj(\text{“}x\text{”})$ and $\vdash_{\mathrm{MT}} \neg Conj(\text{“}x\text{”})$. But this extension would not be in the spirit of a metatheory of a mechanized object theory, in the sense that it would not take into account the fact that the code is partial, e.g. the fact that `CONJ` cannot be run successfully on a data structure recording a variable. Definitions 9.3 and 9.4 capture exactly those expressions that can be evaluated by the simulation structure machinery. Theorem 9.5 captures the actual relation between provability in MT and truth in the mechanizable analogue of its model.

9.3. MT is correct and complete with respect to provability in OT

We prove that tactic terms have the right behaviour.

Theorem 9.6 (MT correct and complete w.r.t. OT). *Let* Π *be a sequent tree of* s. *Let* τ_π *be the tactic term of* Π. *Then*

(1) $\vdash_{\mathrm{MT}} \tau_\pi = \text{“}s\text{”} \iff_{(a)} \Pi$ *is a proof of* $s \iff_{(b)} \vdash_{\mathrm{MT}} T(\tau_\pi)$.
(2) $\vdash_{\mathrm{MT}} \tau_\pi = fail \iff_{(a)} \Pi$ *is not a proof* $\iff_{(b)} \vdash_{\mathrm{MT}} \neg T(\tau_\pi)$.

Part (1) of Theorem 9.6 states that a tactic term corresponds to a successful proof iff it can be proved equal to the name of a sequent (part (1a)) which denotes a theorem of OT (part (1b)). Part (2) states that a tactic term corresponds to a sequent tree which is not a proof iff it can be proved equal to failure (part (2a)) iff it does not denote a theorem of OT (part (2b)). Notice that the fact that a program tactic fails to

prove a sequent does not imply that the sequent is not provable. Analogously, part (2b) of Theorem 9.6 states that the tactic term does not denote a theorem (i.e. $\neg T(\tau_\pi)$), but *does not state* the much stronger fact that s is not a theorem (i.e. $\neg T(\text{“}s\text{”})$). This result is therefore very different from the fact that $\neg T(\text{“}s\text{”})$ is provable in MT iff s is not provable in OT. Theorem 9.6 makes a statement about a single sequent tree Π and not about the provability of s, which would involve considering all sequent trees of s. However, in part (1) of Theorem 9.6, the two notions collapse and from $T(\tau_\pi)$ it is possible to prove $T(\text{“}s\text{”})$. Indeed, as a corollary of Theorem 9.6, we have that if s is provable in OT then $T(\text{“}s\text{”})$ is provable in MT. We have therefore that reflection down and reflection up, namely [22,23]

$$R_{\text{down}}\frac{\vdash_{\text{MT}} T(\text{“}s\text{”})}{\vdash_{\text{OT}} s}, \qquad R_{\text{up}}\frac{\vdash_{\text{OT}} s}{\vdash_{\text{MT}} T(\text{“}s\text{”})} \tag{46}$$

are correct inference rules between theories in the multitheory system MT-OT, and that axioms (11), (12) need not be explicitly and a priori stated. They can in fact be proved and asserted when needed with an application of R_{up}.

9.4. *All tactics are theorems of MT*

We prove that all tactics are theorems of MT.

Theorem 9.7. *Any tactic is provable in MT.*

The fact that all tactics are provable in MT is exactly what we should have expected. In fact, any tactic corresponds to a program tactic which can be defined in the system code. To say that any tactic can be derived in the metatheory is equivalent to saying that any strategy implementing any finite composition of inference steps can be written in HGKM.

In Section 7 we have proved theorems (34), (35) and (36). They represent, without taking into account failure, (derived) object level inference rules. We show now that theorems of this kind can be proved in general. First we need some technical definitions. The *sequent tree term* t_π and the *preconditions* $\mathcal{P}_\pi$ of a sequent tree Π are defined inductively over the structure of sequent trees. In the base case, a sequent tree is a single sequent s. If the sequent is a proof, i.e. it is either an axiom or an assumption, then its sequent tree term is $\text{“}s\text{”}$ and its preconditions are $T(\text{“}s\text{”})$. If it is not a proof, i.e. it is neither an axiom nor an assumption, then its sequent tree term is *fail* and its preconditions are $T(\mathit{fail})$. In the step case, if t_{π_1} and $\mathcal{P}_{\pi_1}$ are the sequent tree term and the preconditions of Π_1, and Π is built from Π_1 by applying $\wedge E_l$ $(\forall I\ a\ x)$ to the end sequent of Π_1, then $\mathit{fandel}(t_{\pi_1})$ $(\mathit{falli}(t_{\pi_1}, \text{“}x\text{”}, \text{“}a\text{”}))$ and $\mathcal{P}_{\pi_1} \wedge \mathit{Conj}(t_{\pi_1})$ $(\mathcal{P}_{\pi_1} \wedge \mathit{Var}(\text{“}x\text{”}) \wedge \mathit{Par}(\text{“}a\text{”}) \wedge \mathit{NoFree}(\text{“}a\text{”}, t_{\pi_1}))$ are the sequent tree term and the preconditions of Π, respectively. For instance, if a sequent tree is built by applying first $\forall E\ x\ a$ to an axiom s, and then $\wedge E_l$ and $\forall I\ x\ a$ are applied in the sequent tree in the given order, then the corresponding sequent tree term is

$$\mathit{falli}(\mathit{fandel}(\mathit{falle}(\text{“}s\text{”}, \text{“}x\text{”}, \text{“}a\text{”})), \text{“}x\text{”}, \text{“}a\text{”})$$

and the corresponding preconditions are

$$T(\text{“}s\text{”})$$
$$\wedge Var(\text{“}x\text{”}) \wedge Term(\text{“}a\text{”}) \wedge Forall(\text{“}s\text{”})$$
$$\wedge Conj(falle(\text{“}s\text{”}, \text{“}x\text{”}, \text{“}a\text{”}))$$
$$\wedge Var(\text{“}x\text{”}) \wedge Par(\text{“}a\text{”}) \wedge NoFree(\text{“}x\text{”}, fandel(falle(\text{“}s\text{”}, \text{“}x\text{”}, \text{“}a\text{”}))).$$

For each OT sequent tree we have a wff of the form $\mathcal{P}_\pi[c_1, \ldots, c_n] \supset T(t_\pi[c_1, \ldots, c_n])$, where $c_1, \ldots, c_n$ are the individual constants appearing in the sequent tree term and in the preconditions. Let $x_1, \ldots, x_n$ be individual variables of MT. We write as $\mathcal{P}_\pi[x_1, \ldots, x_n]$ and $t_\pi[x_1, \ldots, x_n]$ the wff and the term obtained by replacing the constants $c_1, \ldots, c_n$ in $\mathcal{P}_\pi[c_1, \ldots, c_n]$ and $t_\pi[c_1, \ldots, c_n]$ with the variables $x_1, \ldots, x_n$, respectively. We have that:

Theorem 9.8. *Any wff of the form*

$$\forall x_1 \ldots \forall x_n \ (\mathcal{P}_\pi[x_1, \ldots, x_n] \supset T(t_\pi[x_1, \ldots, x_n])) \tag{47}$$

is provable in MT.

9.5. Tactic execution is correct

Under the hypothesis that the underlying implementation is correct, tactic interpretation (described in Section 8.1) is correct, as it is the sequence of four steps, each of which is provably correct. Step 1 is trivially correct. The correctness of Step 2 is a consequence of Theorem 9.6. In fact, if c_i is *fail*, then $\vdash_{MT} Tac(fail)$, while if c_i is “s” with $\vdash_{OT} s$, then we have $\vdash_{MT} T(\text{“}s\text{”})$ and therefore $\vdash_{MT} Tac(\text{“}s\text{”})$. The correctness of Step 3 is a consequence of Theorems 9.5 and 9.6. We compute $\texttt{simplify}(\tau_\pi[c_1, \ldots, c_n])$, where $\tau_\pi[c_1, \ldots, c_n]$ is a simplifiable term. If $\texttt{simplify}(\tau_\pi[c_1, \ldots, c_n]) = s$, then we have that the simplifiable wff $\tau_\pi[c_1, \ldots, c_n] = \text{“}s\text{”}$ is true in $\mathcal{M}$ and, therefore, $\vdash_{MT} \tau_\pi[c_1, \ldots, c_n] = \text{“}s\text{”}$ (Theorem 9.5). From Theorem 9.6 part (1) we have that $\vdash_{MT} T(\text{“}s\text{”})$. If $\texttt{simplify}(\tau_\pi[c_1, \ldots, c_n]) = \texttt{fail}$, then we have that the simplifiable wff $\tau_\pi[c_1, \ldots, c_n] = fail$ is true in $\mathcal{M}$ and, therefore, $\vdash_{MT} \tau_\pi[c_1, \ldots, c_n] = fail$ (Theorem 9.5). From Theorem 9.6 part (2) we have that τ_π does not correspond to a proof. Therefore the interpretation process stops correctly. Finally, property (45) (and also Theorem 9.6) guarantees that Step 4 is correct. The correctness of the interpretation of wffs of the form (47) can be shown exactly in the same way.

The correctness of flattening can be argued very much in the same way, the main difference being that the reflecting up from OT and the assertion of a theorem in OT must be considered in terms of function calls to the HGKM functions `fproof-update` and `TAC` (see Section 5).

9.6. A remark

The theoretical results presented above are all is needed to show that our approach is correct under the hypothesis that the GETFOL code does what it is supposed to do. In fact

we know that MT, as lifted from the code, is consistent, that SIMPLIFY is used correctly, that MT is correct and complete with respect to proofs (and non-proofs) in OT, that it can express and prove all tactics, and that tactics can be executed correctly. We still do not have a guarantee that an incorrect implementation will not derive non-theorems and derive all theorems, not even in principle. In fact we have given all the results with respect to a set-theoretic characterization of what the GETFOL code does, i.e. we have claimed that the code is a finite presentation of the model defined in Section 9.1.

To lift this hypothesis requires axiomatizing the underlying HGKM interpreter. This work is being done as part of the subproject reimplementing GETFOL (see Section 3). Some preliminary results can be found in [1, 18, 20], a full account is the topic of a forthcoming paper. Briefly put, these results start from an axiomatization of the HGKM interpreter, based on the work described in [49, 56]. Then a representability property is shown to hold between the HGKM implementation of OT and MT. This property is similar to the notions of numeralwise representability and numeralwise expressibility, as described for example in [39]. Some complications arise, for instance because we must take into account the fact that GETFOL has state. These results give us the correctness of the GETFOL implementation modulo the correctness of the HGKM interpreter, i.e. modulo the fact that HGKM does what it is supposed to do.

10. Current and future work

As hinted in Section 3, we have an implementation of everything described in this paper. Within GETFOL, we have mechanized OT, MT and the procedures to synthesize and execute tactics. However MT, as described so far, can express a very limited class of tactics, only the tactics that correspond to finite compositions of proof steps. A lot of work is currently under way to overcome the current limitations of MT.

A first step is to extend MT to be expressive enough to represent the program tactics and tacticals used in most tactic-based interactive theorem provers (e.g. [14, 32, 43, 44]). [25] describes how MT can be extended to axiomatize the most interesting tacticals, i.e. *then*, *orelse*, *try*, *progress* and *repeat*. Consider for instance the tactical *repeat*. Its axiomatization is as follows.

$$\begin{aligned}&\forall x \forall t\ (Tac(x) \wedge LTac(t) \supset \\ &\quad apply(repeat(t), x) = \\ &\qquad \textbf{if}\ (apply(t, x) = fail) \\ &\qquad\quad \textbf{then}\ x \\ &\qquad\quad \textbf{else}\ apply(repeat(t), apply(t, x)))\end{aligned}$$

where x and t are individual variables and *LTac* is a unary predicate holding of terms called *logic tactics*. Logic tactics include a constant for each primitive tactic, e.g. "*fandeltac*" and "*fallitac*", and terms constructed by composing logic tactics through tacticals, e.g. *repeat*(*orelse*("*fandeltac*", "*fallitac*")). The function symbol *apply* is used to express tactic application. For instance we have that

$$\forall x\ (Tac(x) \supset apply(\text{"}fandeltac\text{"}, x) = fandeltac(x))$$

is provable in the extended MT. The tactical *repeat* is the standard tactical used to write strategies, based on iteration and on the recursive application of tactics, which do not necessarily correspond to a finite composition of proof steps. This form of recursion can be safely represented. A problem is that, in general, *repeat* is not powerful enough and that, on the other hand, introducing rules which allow for the construction of recursive logic tactics may not preserve consistency. At the moment we are studying some more general sufficient conditions for a characterization of recursive (possibly "not terminating") logic tactics which preserve consistency. We are also studying how to synthesize tactics containing tacticals, extending "naturally" the results presented in Section 9.3.

A second step is to provide MT with induction principles. Induction principles are necessary in order to synthesize or prove the correctness of certain derived inference rules (see for instance [8,37]). Some preliminary experiments of theorem proving in such extensions of MT have been performed. [1] describes a proof of the theorem about formulas containing only equivalences stated in [54] (the same theorem is also stated and proved in [3]). A problem which we are now starting to investigate is how and to what extent such induction principles can be lifted from the code. [20] discusses some ideas about how this can be done for wffs and proofs.

A third step is to extend MT to prove metarules similar to those described in [44]. [52] describes some preliminary work in this direction, limited to the propositional case. These rules are characterized by the fact that they use only wff constructors. Thus, for instance, it is possible to express in our notation the statement "if the formula $A \wedge B$ is a theorem, then A is a theorem" as

$$\forall x_1 \forall x_2 \ (Th(mkand(x_1, x_2)) \supset Th(x_1)),$$

where x_1, x_2 range over wffs, Th is a unary predicate such that $Th(\text{"}A\text{"})$ holds iff $T(\text{"} \longrightarrow A\text{"})$ holds, and *mkand* is a wff constructor which takes two wffs and builds their conjunction. Notice that this is different from what we can prove in MT, i.e.

$$\forall x \ (T(x) \wedge Conj(x) \supset T(fandel(x))),$$

where x ranges over sequents and *fandel* is, intuitively speaking, a proof constructor. This extension would allow us to extract from any given tactic which explicitly represents all the proofs steps, a corresponding new tactic of only one proof step (and which does all the manipulation at the formula level). This latter tactic corresponds to a program tactic that is in general much faster to execute. An interesting open problem here is whether it is possible to do this by using (an extended version of) the simulation structure machinery, and therefore, without explicitly adding axioms to MT.

Finally we have just started to investigate how to use MT to synthesize interesting tactics effectively. We have started to develop a set of rewriting functions similar to those implemented in Cambridge LCF and described in [45]. Our goal is to perform in MT a similar kind of reasoning to that performed in proof planning [10].

11. Related work

Compared to the previous research in metalevel theorem proving, the work described in this paper is limited in at least three respects. First, it does not allow the use of expressive control structures and tacticals, as it is the case, for instance in [16,30,37] (but this has been fixed in [25]). Second, it does not allow induction, which is used for instance in the metatheories described in [8,37]. Third, so far the system has been used only to synthesize simple tactics. As described in Section 10, these topics are currently being investigated. We actually hope that the techniques elaborated in the past will largely apply to MT and its extensions (the extent to which this is true is still to be found out). This investigation promises to be a very interesting project as trying to keep the connection between the metatheory and the code may lead us to see the previous work (and in particular ours) in a new perspective. Finally, MT is first order, unlike in the work described in [16,37,40,44]. This provides some advantages, see for instance the discussions in [12,29], however it might prevent us from performing interesting forms of reasoning.

However, these issues, though very important, are somehow orthogonal to the main message of this paper, which is about describing a metatheory of a mechanized object theory, i.e. a metatheory which can be put in correspondence with the code implementing the object theory, and about how this can be exploited to perform lifting, flattening and tactic interpretation. None of the metatheories developed in the past (besides, of course, the work on FOL, see Section 3) has features similar to MT. This is the case, for instance for LCF [32], NuPRL [14,37,40], Isabelle [44], HOL [31], OBJ3 [29,30], for the provers based on logic programming (see e.g. [5,6,16,35,36]) and for the logical frameworks [2,3,33]. This has some consequences. None of these systems can reason about the underlying code. Even if LCF, HOL, NuPRL and Isabelle provide a metalanguage for writing program tactics, there seems to be no straightforward way to translate them into metalevel logical statements or vice versa. In the area of logic programming, even if they can control the Prolog search strategy, the metainterpreters cannot modify it. That is, the user can write a metainterpreter for any desired search strategy, however the metainterpreter will be executed by using the Prolog built-in search strategy.

The fact that MT is a metatheory of a mechanized object theory gives it some features which make it somewhat unusual. Thus, for instance, some of the features of MT that cannot be found in any of the work described in the past are the following: the syntax is not explicitly axiomatized and the needed facts are extracted from the code using the simulation structure machinery, in the axiomatization of the syntax only ground facts are considered; the notion of failure is explicitly axiomatized, using `fail` and FAIL, and kept distinct form the notion of partialness; MT makes a distinction between inference rules and primitive tactics and has a notion of tactic. Some more usual features, but still not standard are the following: inference rules are functions and not predicates, as it happens for instance in [53]; inference rules do not take theories and signatures as arguments, as it happens for instance in [53] (this in GETFOL is solved using the multicontext machinery [17]); even if MT can reason about proofs this notion is not explicitly axiomatized, as it happens for instance in [3,53].

In our approach the code implementing inference rules and program tactics is really like axioms, i.e. modulo lifting, this code can be added to MT and used to derive new facts. In this perspective our work is similar in spirit to Boyer and Moore's work on metafunctions [8] (modulo the limitations described at the beginning of this section). A difference is that in the work described in this paper (also considering the extensions allowing tacticals) we have mainly considered how to compose simple tactics into more complex tactics. In [8] the emphasis is instead on proving the correctness of derived inference rules (not expressed as composition of simpler inference rules) using induction principles.

Our long term goal, far from being achieved, is to develop GETFOL into a system whose code is provably correct, and which provides facilities for provably correct system development. This goal is similar to that underlying the development of Acl2 [9], a reimplementation of a portion of the Boyer and Moore theorem prover [7], using the same logic for which Acl2 is a theorem prover. One main difference is that in Acl2 the logic language and the implementation language are the same. This is possible since Acl2 is written applicatively. On the other hand, GETFOL has a lot of state, e.g. the language of a theory, the axioms, the theorems and the proofs constructed so far, but also global variables used to optimize the implementation of decision procedures, counters used for the automatic generation of different names for skolem functions, and so on. This gives us some advantages, like that of being able of showing the proof constructed so far; however it complicates the relation between the implementation and the logical language (they are essentially identical only for what concerns the computation machinery, which is in fact functional, see Sections 5 and 6). Some preliminary discussions about how to hide state during the lifting are done in [18,20]. Some hints are also in Section 6, in the specific case of lifting the update and top level machinery implementing GETFOL proofs.

We share the goal of self-reflection with a lot of work in the programming language community (see for instance [38,57]), one of the first contributions in this area being the work on 3-lisp [47]. The substantial difference is that in our approach the introspection is performed by deduction instead of by computation.

Finally, as a minor remark, a further difference with a lot of the related work, with the noticeable exceptions of [10,36,55], is that MT is distinct from OT. (Some motivations and advantages for this choice are given in [22].) In Gödel [36], in particular, the naming relation has some commonalities with that employed in MT. Roughly speaking, both MT and Gödel allow for structural descriptive names. One difference is that GETFOL has no hardwired naming machinery and that the objects of OT can be given arbitrary names.

12. Conclusion and acknowledgements

The work described in this paper is part of a long term project whose final goal is to build GETFOL into a self-reflective system able to introspect, reason about, extend and modify its own code. This work started in 1988 when the first author was at the AI Department of Edinburgh University. In Edinburgh, financial support for the first author

was provided by SERC grant GR/E/4459.8. Currently the first and second author's research at IRST is funded by ITC (Istituto Trentino di Cultura). At the moment the project is being developed within the Mechanized Reasoning Group(s) (MRG) at IRST and DIST (Department of Communcation, Computer and System Sciences, University of Genoa). Paolo Pecchiari (DIST) has completely reimplemented the simulation structure machinery and the FOL rewriter. Alessandro Armando (DIST), Alessandro Cimatti (IRST), Michela Della Lucia (IRST), Luca Vigano' (DIST, currently Max-Planck Institute, Saarbruecken) and Alessandro Zorer (IRST) have worked on the project whose goal is the design and reimplementation of GETFOL. Massimo Benerecetti (IRST) has mechanized MT (extended to allow the use of tacticals) in GETFOL. Without the work of these people, and equally important, without their continuous feedback, the work described in this paper could have never been done. This work has been motivated and strongly influenced by the collaboration and discussions with Alan Bundy and Richard Weyhrauch. David Basin, Frank Van Harmelen, John McCarthy, Luciano Serafini, Alex K. Simpson, Alan Smaill, Carolyn Talcott and Toby Walsh have provided useful feedback on various aspects of the work described in this paper. We thank Toby Walsh for carefully proof reading the paper. Finally, the feedback provided by the referees has helped us to improve substantially the quality of the presentation.

Appendix A. The metatheory MT

MT is a triple MT = $\langle \mathcal{ML}, \mathcal{MA}x, \mathcal{MR} \rangle$ where $\mathcal{ML}$, $\mathcal{MA}x$ and $\mathcal{MR}$ are the language, the set of axioms and the set of inference rules of MT, respectively.

A.1. The language $\mathcal{ML}$

Individual constants

The quotation mark names of the objects of OT plus *fail*.

Function symbols

$\wedge I$: *fandi*,*fanditac* of arity 2,

$\wedge E_l$: *fandel*,*fandeltac* of arity 1,

$\wedge E_r$: *fander*,*fandertac* of arity 1,

$\supset I$: *fimpi*,*fimpitac* of arity 2,

$\supset E$: *fimpe*,*fimpetac* of arity 2,

$\forall I$: *falli*,*fallitac* of arity 3,

$\forall E$: *falle*,*falletac* of arity 3,

$\perp_c$: *false*,*falsetac* of arity 2.

Predicate symbols

$Par, Var, Term, Sec, Wff$ of arity 1,

$Conj, Imp, Forall, False$ of arity 1,

$T, Fail, Tac$ of arity 1,

$NoFree, Hp$ of arity 2,

$=$ of arity 2.

Sentential constants

$\bot, \top$.

A.2. The axioms $\mathcal{MA}x$

Basic axioms

$$\top$$
$$\forall x\ \neg(Sec(x) \wedge Fail(x))$$
$$\forall x\ (T(x) \supset Sec(x))$$
$$\forall x\ (Fail(x) \leftrightarrow x = fail)$$

Inference rule axioms

$$\forall x_1 \forall x_2\ \neg\, fandi(x_1, x_2) = fail,$$
$$\forall x\ \neg\, fandel(x) = fail,$$
$$\forall x\ \neg\, fander(x) = fail,$$
$$\forall x_1 \forall x_2\ \neg\, fimpi(x_1, x_2) = fail,$$
$$\forall x_1 \forall x_2\ \neg\, fimpe(x_1, x_2) = fail,$$
$$\forall x_1 \forall x_2 \forall x_3\ \neg\, falli(x_1, x_2, x_3) = fail,$$
$$\forall x_1 \forall x_2 \forall x_3\ \neg\, falle(x_1, x_2, x_3) = fail,$$
$$\forall x_1 \forall x_2\ \neg\, false(x_1, x_2) = fail.$$

Computation machinery axioms

$$\forall x_1 \forall x_2\ (Tac(x_1) \wedge Tac(x_2) \supset$$
$$fanditac(x_1, x_2) = \textbf{if}\ \neg Fail(x_1) \wedge \neg Fail(x_2)\ \textbf{then}\ fandi(x_1, x_2)\ \textbf{else}\ fail)$$

$$\forall x\ (Tac(x) \supset$$
$$fandeltac(x) = \textbf{if}\ \neg Fail(x) \wedge Conj(x)\ \textbf{then}\ fandel(x)\ \textbf{else}\ fail)$$

$$\forall x\ (Tac(x) \supset$$
$$fandertac(x) = \textbf{if}\ \neg Fail(x) \wedge Conj(x)\ \textbf{then}\ fander(x)\ \textbf{else}\ fail)$$

$$\forall x_1 \forall x_2\ (Tac(x_2) \supset$$
$$fimpitac(x_1, x_2) = \textbf{if}\ \neg Fail(x_2) \wedge Wff(x_1)\ \textbf{then}\ fimpi(x_1, x_2)\ \textbf{else}\ fail)$$

$$\forall x_1 \forall x_2\ (Tac(x_1) \wedge Tac(x_2) \supset$$
$$fimpetac(x_1, x_2) = \textbf{if } \neg Fail(x_1) \wedge \neg Fail(x_2) \wedge Imp(x_2) \wedge Hp(x_1, x_2)$$
$$\textbf{then } fimpe(x_1, x_2) \textbf{ else } fail)$$

$$\forall x_1 \forall x_2 \forall x_3\ (Tac(x_1) \supset$$
$$fallitac(x_1, x_2, x_3)$$
$$= \textbf{if } \neg Fail(x_1) \wedge Var(x_2) \wedge Par(x_3) \wedge NoFree(x_3, x_1)$$
$$\textbf{then } falli(x_1, x_2, x_3) \textbf{ else } fail)$$

$$\forall x_1 \forall x_2 \forall x_3\ (Tac(x_1) \supset$$
$$falletac(x_1, x_2, x_3)$$
$$= \textbf{if } \neg Fail(x_1) \wedge Var(x_2) \wedge Term(x_3) \wedge Forall(x_1)$$
$$\textbf{then } falle(x_1, x_2, x_3) \textbf{ else } fail)$$

$$\forall x_1 \forall x_2\ (Tac(x_1) \supset$$
$$falsetac(x_1, x_2) = \textbf{if } \neg Fail(x_1) \wedge Wff(x_2) \wedge False(x_1)$$
$$\textbf{then } false(x_1, x_2) \textbf{ else } fail)$$

Update machinery axioms

$$\forall x\ (Tac(x) \leftrightarrow T(x) \vee Fail(x)).$$

Top level machinery axioms

$$\forall x_1 \forall x_2\ (Tac(x_1) \wedge Tac(x_2) \supset Tac(fanditac(x_1, x_2)))$$
$$\forall x\ (Tac(x) \supset Tac(fandeltac(x)))$$
$$\forall x\ (Tac(x) \supset Tac(fandertac(x)))$$
$$\forall x_1 \forall x_2\ (Tac(x_2) \supset Tac(fimpitac(x_1, x_2)))$$
$$\forall x_1 \forall x_2\ (Tac(x_1) \wedge Tac(x_2) \supset Tac(fimpetac(x_1, x_2)))$$
$$\forall x_1 \forall x_2 \forall x_3\ (Tac(x_1) \supset Tac(fallitac(x_1, x_2, x_3)))$$
$$\forall x_1 \forall x_2 \forall x_3\ (Tac(x_1) \supset Tac(falletac(x_1, x_2, x_3)))$$
$$\forall x_1 \forall x_2\ (Tac(x_1) \supset Tac(falsetac(x_1, x_2)))$$

Let a, x, t, and w be any individual parameter, individual variable, term and wff of OT. Let A, B and C be wffs of OT. Let Γ and Δ be finite sets of formulas of OT. Let c, c_1, c_2 be constants of OT. Let ξ be any object of OT.

Ground axioms about T

$$T(\text{“}A \longrightarrow A\text{”})$$
$$T(\text{“} \longrightarrow A\text{”}), \quad \text{if } \longrightarrow A \in \mathcal{A}x$$

Ground axioms about inference rules

$fandi(\text{“}\Gamma \longrightarrow A\text{”}, \text{“}\Delta \longrightarrow B) = \text{“}\Gamma, \Delta \longrightarrow A \wedge B\text{”}$
$fandel(\text{“}\Gamma \longrightarrow A \wedge B\text{”}) = \text{“}\Gamma \longrightarrow A\text{”}$
$fander(\text{“}\Gamma \longrightarrow A \wedge B\text{”}) = \text{“}\Gamma \longrightarrow B\text{”}$
$fimpi(\text{“}A\text{”}, \text{“}\Gamma \longrightarrow B\text{”}) = \text{“}\Gamma - \{A\} \longrightarrow A \supset B\text{”}$
$fimpe(\text{“}\Gamma \longrightarrow A\text{”}, \text{“}\Delta \longrightarrow A \supset B) = \text{“}\Gamma, \Delta \longrightarrow B\text{”}$
$falli(\text{“}\Gamma \longrightarrow A\text{”}, \text{“}x\text{”}, \text{“}a\text{”}) = \text{“}\Gamma \longrightarrow \forall x A_x^a\text{”}$, where a does not occur in Γ
$falle(\text{“}\Gamma \longrightarrow \forall x A\text{”}, \text{“}x\text{”}, \text{“}t\text{”}) = \text{“}\Gamma \longrightarrow A_x^t\text{”}$
$fimpi(\text{“}\Gamma \longrightarrow \bot\text{”}, \text{“}A\text{”}) = \text{“}\Gamma - \{A \supset \bot\} \longrightarrow A\text{”}$

Ground axioms about syntax

$Par(\text{“}a\text{”})$
$\neg Par(c)$, if c is *fail* or “ξ” and ξ is not an individual parameter of OT
$Var(\text{“}x\text{”})$
$\neg Var(c)$, if c is *fail* or “ξ” and ξ is not an individual variable of OT
$Term(\text{“}t\text{”})$
$\neg Term(c)$, if c is *fail* or “ξ” and ξ is not a term of OT
$Wff(\text{“}w\text{”})$
$\neg Wff(c)$, if c is *fail* or “ξ” and ξ is not a wff of OT
$\neg\, c_1 = c_2$, if c_1 and c_2 are distinct individual constants
$Conj(\text{“}\Gamma \longrightarrow A \wedge B\text{”})$
$\neg Conj(\text{“}\Gamma \longrightarrow A\text{”})$, if A is not a conjunction
$Imp(\text{“}\Gamma \longrightarrow A \supset B\text{”})$
$\neg Imp(\text{“}\Gamma \longrightarrow A\text{”})$, if A is not an implication
$Hp(\text{“}\Gamma \longrightarrow A\text{”}, \text{“}\Delta \longrightarrow A \supset B\text{”})$
$\neg Hp(\text{“}\Gamma \longrightarrow A\text{”}, \text{“}\Delta \longrightarrow C\text{”})$, if C is not of the form $A \supset B$
$NoFree(\text{“}a\text{”}, \text{“}\Gamma \longrightarrow A\text{”})$, if a does not appear in Γ
$\neg NoFree(\text{“}a\text{”}, \text{“}\Gamma \longrightarrow A\text{”})$, if a appears in Γ
$Forall(\text{“}\Gamma \longrightarrow \forall x A\text{”})$
$\neg Forall(\text{“}\Gamma \longrightarrow A\text{”})$, if A is not universally quantified
$False(\text{“}\Gamma \longrightarrow \bot\text{”})$
$\neg False(\text{“}\Gamma \longrightarrow A\text{”})$, if A is not $\bot$
$Sec(\text{“}\Gamma \longrightarrow A\text{”})$
$\neg Sec(\text{“}\xi\text{”})$, if ξ is not a sequent of OT

Appendix B. Proofs

B.1. Proof of Theorem 9.2

We show that for any wff α in $\mathcal{ML}$, $\vdash_{MT} \alpha$ implies $\models_{\mathcal{M}} \alpha$. Axiom (1) is true since $g(Sec) \cap g(Fail) = \emptyset$. Axiom (2) is true since $g(T) \subseteq g(Sec)$. Axiom (3) defines the predicate *Fail*. Axioms (4) and (5) are true since $g(fandel)(d) \notin \{\mathrm{F}\}$ and $g(falli)(d_1, d_2, d_3) \notin \{\mathrm{F}\}$ for any $d, d_1, d_2, d_3 \in \mathcal{D}$. Consider axiom (6). Let x_1 be

assigned to $d \in g(Tac)$. If $d \in g(Conj)$, then the conditional term is interpreted into $g(fandel)(d)$. If $d \notin g(Conj)$, then it is interpreted into F. Then axiom (6) is true. The proof for axiom (7) is analogous. Axiom (8) defines the predicate *Tac*. Axiom (9) is true since, if x is assigned to $d \in g(Tac)$, then $g(fandeltac)(d) \in g(Tac)$. The proof for axiom (10) is analogous. Axioms (11), (12) are true since OT axioms and assumptions belong to $g(T)$. Axioms (13), (14) are true since $g(fandel)(\Gamma \longrightarrow A \wedge B) = \Gamma \longrightarrow A$ and, if a does not appear in Γ, $g(falli)(\Gamma \longrightarrow A, x, a) = \Gamma \longrightarrow \forall x A_x^a$. Axioms (15)–(25) are trivially true.

B.2. Proof of Theorem 9.5

We first prove the following lemma.

Lemma B.1. *Let $t \in S_t$. Let $c \in S_t$ be a constant of MT. If $\models_{\mathcal{M}} t = c$, then $\vdash_{\mathrm{MT}} t = c$.*

Proof. By induction over the structure of t.

Base. Obvious since we have $c = c$.

Step. We have one case for each form of simplifiable term (Definition 9.3). Let $t, t_1, t_2, t_3 \in S_t$.

(1) *fandel*(t). t denotes a sequent of the form $\Gamma \longrightarrow A \wedge B$, since $t \in S_t$. Then $\models_{\mathcal{M}} t =$ "$\Gamma \longrightarrow A \wedge B$". From the induction hypotheses we have that $\vdash_{\mathrm{MT}} t =$ "$\Gamma \longrightarrow A \wedge B$". $\models_{\mathcal{M}}$ *fandel*$(t) = c$ implies that c denotes $\Gamma \longrightarrow A$. *fandel*("$\Gamma \longrightarrow A \wedge B$") = "$\Gamma \longrightarrow A$" is axiom (13).

(2) *falli*(t_1, t_2, t_3). We have that t_1 denotes a sequent of the form $\Gamma \longrightarrow A$, t_2 denotes an individual variable x and t_3 denotes an individual parameter a which does not appear free in Γ. From the induction hypotheses we have that $\vdash_{\mathrm{MT}} t_1 =$ "$\Gamma \longrightarrow A$", $\vdash_{\mathrm{MT}} t_2 =$ "x" and $\vdash_{\mathrm{MT}} t_3 =$ "a". c denotes $\Gamma \longrightarrow \forall x A_x^a$. *falli*("$\Gamma \longrightarrow A$", "$x$", "$a$") = "$\Gamma \longrightarrow \forall x A_x^a$" is axiom (14).

(3) *fandeltac*(t). The proof is similar to the proof for *fandel*(t). If t denotes a sequent that is a conjunction, then we use axioms (6) and (20). If t denotes a sequent that is not a conjunction, then we use axioms (6) and (21). If t denotes F, then we use axioms (6) and (3).

(4) *fallitac*(t_1, t_2, t_3). The proof is similar to the proof for *falli*(t_1, t_2, t_3). □

Now we prove the following theorem.

Theorem B.2. *Let $w \in S_w$. Then $\models_{\mathcal{M}} w \Longrightarrow \vdash_{\mathrm{MT}} w$.*

Proof. We have one case for each form of simplifiable wff (Definition 9.4).

(1) $t_1 = t_2$. By induction over t_2. The base case is Lemma B.1. The step cases are analogous to the step cases of the proof of Lemma B.1.

(2) *Sec*(t). t denotes a sequent s. From Lemma B.1 we have $\vdash_{\mathrm{MT}} t =$ "s". *Sec*("s") is axiom (24).

(3) *Fail*(t). t denotes F. From Lemma B.1 we have $\vdash_{\mathrm{MT}} t =$ *fail*. From definition (3) we have $\vdash_{\mathrm{MT}}$ *Fail*(*fail*).

(4) *Conj*(t). t denotes a sequent s whose formula is a conjunction. From Lemma B.1 we have $\vdash_{MT} t =$ "s". *Conj*("s") is axiom (20).
(5) *Par*(t). t denotes an individual parameter a. From Lemma B.1 we have $\vdash_{MT} t =$ "a". *Par*("a") is axiom (15).
(6) *Var*(t). t denotes an individual variable x. From Lemma B.1 we have $\vdash_{MT} t =$ "x". *Var*("x") is axiom (17).
(7) *Nofree*(t_1, t_2). t_2 denotes a sequent $\Gamma \longrightarrow A$ and t_1 denotes an individual parameter a that does not appear in Γ. From Lemma B.1 we have $\vdash_{MT} t_2 =$ "$\Gamma \longrightarrow A$" and $\vdash_{MT} t_1 =$ "a". *Nofree*("a", "$\Gamma \longrightarrow A$") is axiom (22). □

We prove the following lemma.

Lemma B.3. *Let $t \in S_t$. Let $c \in S_t$ be a constant of MT. If $\not\models_{\mathcal{M}} t = c$, then $\vdash_{MT} \neg\, t = c$.*

Proof. We have one case for each form of simplifiable term (Definition 9.3). If $\not\models_{\mathcal{M}} c_1 = c_2$, then c_1 and c_2 are distinct constants. $\neg\, c_1 = c_2$ is axiom (19). Consider now simplifiable terms of the form *fandel*(t). From the fact that $t \in S_t$ we have that *fandel*(t) denotes a sequent s and, therefore, $\vdash_{MT}$ *fandel*(t) = "s" (Theorem B.2). $\not\models_{\mathcal{M}}$ *fandel*(t) = c implies that $\not\models_{\mathcal{M}}$ "s" = c and, therefore, $\vdash_{MT} \neg$"s" = c (axiom (19)). The proof for the other cases is similar. □

Finally, we prove the following theorem.

Theorem B.4. *Let $w \in S_w$. Then $\not\models_{\mathcal{M}} w \Longrightarrow \vdash_{MT} \neg w$.*

Proof. We have one case for each form of simplifiable wff (Definition 9.4).
(1) $t_1 = t_2$. By induction over t_2. The base case is Lemma B.3. The step cases are analogous to the step cases of the proof of Lemma B.3.
(2) *Sec*(t). Either t denotes any object ξ of OT that is not a sequent or it denotes F. In the former case, from Theorem B.2 we have $\vdash_{MT} t =$ "ξ". $\neg$*Sec*("ξ") is axiom (25). In the latter case, from Theorem B.2 we have $\vdash_{MT} t =$ *fail*. From axiom (1) we have $\vdash_{MT} \neg$*Sec*(t).
(3) *Fail*(t). t denotes a sequent s. From Theorem B.2 we have $\vdash_{MT} t =$ "s". From axiom (1) we have $\vdash_{MT} \neg$*Fail*(t).
(4) *Conj*(t). t denotes a sequent s whose formula is not a conjunction. From Theorem B.2 we have $\vdash_{MT} t =$ "s". $\neg$*Conj*("s") is axiom (21).
(5) *Par*(t). t denotes either F or an object ξ of OT which is not an individual parameter. From Theorem B.2 we have either $\vdash_{MT} t =$ *fail* or $\vdash_{MT} t =$ "ξ". From axiom (16) we have either $\vdash_{MT} \neg$*Par*(*fail*) or $\vdash_{MT} \neg$*Par*("ξ").
(6) *Var*(t). Proof analogous to the proof for *Par*(t).
(7) *Nofree*(t_1, t_2). t_2 denotes a sequent $\Gamma \longrightarrow A$ and t_1 denotes an individual parameter a that appears in Γ. From Theorem B.2 we have $\vdash_{MT} t_2 =$ "$\Gamma \longrightarrow A$" and $\vdash_{MT} t_1 =$ "a". $\neg$*Nofree*("a", "$\Gamma \longrightarrow A$") is axiom (23). □

B.3. Proof of Theorem 9.6

Theorem B.5 proves parts (1a) $\Leftarrow$ and (1b) $\Rightarrow$, Theorem B.6 proves part (2a) $\Leftarrow$ and (2b) $\Rightarrow$. In the proofs, we call Π the sequent tree of s built by applying an inference rule to the sequent s_1, end sequent of Π_1. We call τ_{π_1} and τ_π the tactic terms of Π_1 and Π, respectively. The proofs are by induction over the structure of sequent trees. We give only the proof for $(\forall I)$ as conceptually identical to that for $(\wedge E_l)$. (1a) $\Rightarrow$, (1b) $\Leftarrow$, (2a) $\Rightarrow$ and (2b) $\Leftarrow$ are trivial corollaries of Theorems B.5 and B.6 and Theorem 9.2 (proofs by contradiction).

Theorem B.5. *Let Π be a sequent tree of s. Let τ_π be the tactic term of Π. If Π is a proof of s, then* $\vdash_{\mathrm{MT}} \tau_\pi = \text{“}s\text{”}$ *and* $\vdash_{\mathrm{MT}} T(\text{“}s\text{”})$.

Proof.

Base. If Π is s, then it must be either an axiom or an assumption. Then τ_π is “s” and $T(\text{“}s\text{”})$ is either axiom (11) or axiom (12).

Step. τ_π is $fallitac(\tau_{\pi_1}, \text{“}x\text{”}, \text{“}a\text{”})$. From the induction hypotheses $\vdash_{\mathrm{MT}} \tau_{\pi_1} = \text{“}s_1\text{”}$ and $\vdash_{\mathrm{MT}} T(\text{“}s_1\text{”})$. From $\vdash_{\mathrm{MT}} T(\text{“}s_1\text{”})$ we have $\vdash_{\mathrm{MT}} Tac(\text{“}s_1\text{”})$. From axiom (7) we have

$$\begin{aligned}&\vdash_{\mathrm{MT}} fallitac(\tau_{\pi_1}, \text{“}x\text{”}, \text{“}a\text{”})\\ &\quad = \textbf{if } \neg Fail(\text{“}s_1\text{”}) \wedge Var(\text{“}x\text{”}) \wedge Par(\text{“}a\text{”}) \wedge NoFree(\text{“}a\text{”}, \text{“}s_1\text{”})\\ &\qquad \textbf{then } falli(\text{“}s_1\text{”}, \text{“}x\text{”}, \text{“}a\text{”})\\ &\qquad \textbf{else } fail\end{aligned} \tag{B.1}$$

From axiom (1) and ground axioms we have $\vdash_{\mathrm{MT}} \tau_\pi = falli(\text{“}s_1\text{”}, \text{“}x\text{”}, \text{“}a\text{”})$ and $\vdash_{\mathrm{MT}} \tau_\pi = \text{“}s\text{”}$. From axiom (10) we obtain $\vdash_{\mathrm{MT}} Tac(fallitac(\text{“}s_1\text{”}, \text{“}x\text{”}, \text{“}a\text{”}))$ and therefore $\vdash_{\mathrm{MT}} Tac(\text{“}s\text{”})$. From axiom (1) we have $\vdash_{\mathrm{MT}} T(\text{“}s\text{”})$. □

Theorem B.6. *Let Π be a sequent tree of s. Let τ_π be the tactic term of Π. If Π is not a proof, then* $\vdash_{\mathrm{MT}} \tau_\pi = fail$ *and* $\vdash_{\mathrm{MT}} \neg T(\tau_\pi)$.

Proof.

Base. If Π is s, then it is neither an axiom nor an assumption. Then τ_π is *fail*.

Step. τ_π is $fallitac(\tau_{\pi_1}, \text{“}x\text{”}, \text{“}a\text{”})$. We have two cases.

(1) Π_1 is a proof. From Theorem B.5 we have that $\vdash_{\mathrm{MT}} \tau_{\pi_1} = \text{“}s_1\text{”}$ and $\vdash_{\mathrm{MT}} T(\text{“}s_1\text{”})$, which implies $\vdash_{\mathrm{MT}} Tac(\text{“}s_1\text{”})$. From axiom (7) we have (B.1). From axiom (23) we have $\vdash_{\mathrm{MT}} \tau_\pi = fail$ and therefore $\vdash_{\mathrm{MT}} \neg T(\tau_\pi)$ (axioms (1), (2)).

(2) Π_1 is not a proof. From the induction hypotheses we have that $\vdash_{\mathrm{MT}} \tau_{\pi_1} = fail$. Therefore $\vdash_{\mathrm{MT}} Tac(\tau_{\pi_1})$. From axiom (7) we have

$$\begin{aligned}\vdash_{\mathrm{MT}} \tau_\pi = \textbf{if } &\neg Fail(fail) \wedge Var(\text{“}x\text{”}) \wedge Par(\text{“}a\text{”}) \wedge NoFree(\text{“}a\text{”}, fail)\\ &\textbf{then } falli(fail, \text{“}x\text{”}, \text{“}a\text{”})\\ &\textbf{else } fail\end{aligned}$$

Since $\vdash_{\mathrm{MT}} Fail(fail)$, we prove $\tau_\pi = fail$ and therefore $\vdash_{\mathrm{MT}} \neg T(\tau_\pi)$. □

B.4. Proof of Theorem 9.7

Proof by induction over the structure of $\tau_\pi(x_1,\ldots,x_n)$.

Base. In the base case a tactic term is either "s" or *fail*. The corresponding tactic is $\forall x\ (Tac(x) \supset Tac(x))$.

Step. We consider only the case of $(\forall I)$ as the case of $(\wedge E_l)$ is very similar. Induction hypotheses are

$$\vdash_{\mathrm{MT}}\ \forall x_1 \ldots \forall x_n\ (Tac(x_1) \wedge \cdots \wedge Tac(x_m) \supset Tac(\tau_\pi[x_1,\ldots,x_n]))$$

and we have to prove

$$\begin{aligned}\vdash_{\mathrm{MT}}\ & \forall x_1,\ldots,\forall x_n,\forall x_{n+1},\forall x_{n+2}\\ & (Tac(x_1) \wedge \cdots \wedge Tac(x_m) \supset\\ & \quad Tac(fallitac(\tau_\pi[x_1,\ldots,x_n],x_{n+1},x_{n+2}))).\end{aligned} \tag{B.2}$$

Let $a_1,\ldots,a_{n+2}$ be individual parameters of MT. From axiom (10) we prove in MT

$$Tac(\tau_\pi[a_1,\ldots,a_n]) \supset Tac(fallitac(\tau_\pi[a_1,\ldots,a_n],a_{n+1},a_{n+2})) \tag{B.3}$$

From the induction hypotheses and (B.3) we prove

$$Tac(a_1) \wedge \cdots \wedge Tac(a_m) \supset Tac(fallitac(\tau_\pi[a_1,\ldots,a_n],a_{n+1},a_{n+2})). \tag{B.4}$$

We apply $\forall I$ to (B.4) and prove (B.2).

B.5. Proof of Theorem 9.8

$\mathcal{P}_\pi$ and t_π are defined inductively over the structure of sequent trees. In the base case, we have $\mathcal{P}_\pi$ and t_π such that (47) is $\forall x\ (T(x) \supset T(x))$. Consider now the step case. We write (47) in the following form.

$$\forall x_1 \ldots \forall x_n\quad (T(x_1) \wedge \cdots \wedge T(x_m) \wedge \mathcal{P}^*_\pi[x_1,\ldots,x_n] \supset T(t_\pi[x_1,\ldots,x_n])) \tag{B.5}$$

where $\mathcal{P}^*_\pi[x_1,\ldots,x_n]$ does not contain occurrences of T. We assume the hypotheses of (B.5) and derive $Tac(x_1) \wedge \cdots \wedge Tac(x_m)$. From (29) we obtain $Tac(\tau_\pi[x_1,\ldots,x_n])$. We can easily derive

$$\begin{aligned}\forall x_1 \ldots \forall x_n\quad & \tau_\pi[x_1,\ldots,x_n]\\ & = \textbf{if } \neg Fail(x_1) \wedge \cdots \wedge \neg Fail(x_m) \wedge \mathcal{P}^*_\pi[x_1,\ldots,x_n]\\ & \quad \textbf{then } t_\pi[x_1,\ldots,x_n]\\ & \quad \textbf{else } fail\end{aligned} \tag{B.6}$$

From the assumption we derive $\neg Fail(x_1) \wedge \cdots \wedge \neg Fail(x_m) \wedge \mathcal{P}^*_\pi[x_1,\ldots,x_n]$. Therefore we obtain $\tau_\pi[x_1,\ldots,x_n] = t_\pi[x_1,\ldots,x_n]$ and thus $Tac(t_\pi[x_1,\ldots,x_n])$, which is equivalent to $T(t_\pi[x_1,\ldots,x_n]) \vee t_\pi[x_1,\ldots,x_n] = fail$ (definition (8)). From axioms (4), (5) we easily prove that $\neg t_\pi[x_1,\ldots,x_n] = fail$ and therefore $T(t_\pi[x_1,\ldots,x_n])$.

References

[1] A. Armando, Architetture riflessive per la deduzione automatica, Ph.D. Thesis, DIST, University of Genoa, Genoa (1993).

[2] A. Avron, F. Honsell and I. Mason, Using typed lambda calculus to implement formal systems on a machine, LFCS Report Series ECS-LFCS-89-72, Laboratory for the Foundations of Computer Science, Computer Science Department, University of Edinburgh, Edinburgh (1989).

[3] D. Basin and R. Constable, Metalogical frameworks, in: *Proceedings Second Workshop on Logical Frameworks*, Edinburgh (1991).

[4] D. Basin, F. Giunchiglia and P. Traverso, Automating meta-theory creation and system extension, in: *AI*IA 1991, 2nd Conference of the Italian Association for Artificial intelligence* (Springer, Berlin, 1991) 48–57; also: IRST-Technical Report 9101-04, IRST, Trento.

[5] K.A. Bowen and R.A. Kowalski, Amalgamating language and meta-language in logic programming, in: S. Tarlund, ed., *Logic Programming* (Academic Press, New York, 1982) 153–173.

[6] K.A. Bowen and T. Weiberhg, A meta-level extension of prolog, in: *IEEE Symposium on Logic Programming*, Boston, MA (1985) 669–675.

[7] R.S. Boyer and J.S. Moore, *A Computational Logic* (Academic Press, New york, 1979).

[8] R.S. Boyer and J.S. Moore, Metafunctions: proving them correct and using them efficiently as new proof procedures, in: R.S. Boyer and J.S. Moore, eds., *The Correctness Problem in Computer Science* (Academic Press, New York, 1981) 103–184.

[9] R.S. Boyer and J.S. Moore, A theorem prover for a computational logic, in: *Proceedings 10th Conference on Automated Deduction*, Lecture Notes in Computer Science 449 (Springer, Berlin, 1990) 1–15.

[10] A. Bundy, The use of explicit plans to guide inductive proofs, in: R. Luck and R. Overbeek, eds., *Proceedings 9th Conference on Automated Deduction* (Springer, Berlin, 1988) 111–120; Longer version available as DAI Research Paper No. 349, Department of Artificial Intelligence, Edinburgh.

[11] A. Bundy and B. Welham, Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation, *Artif. Intell.* **16** (1981) 189–212; also: DAI Research Paper 121, Department of Artificial Intelligence, Edinburgh.

[12] R. Cartwright and J. McCarthy, Recursive programs as functions in a first order theory, SAIL MEMO AIM-324 (1979); also: CS Department Report STAN-CS-79-17.

[13] C.C. Chang and J.M. Keisler, *Model Theory* (North-Holland, Amsterdam, 1973).

[14] R.L. Constable, S.F. Allen and H.M. Bromley et al., *Implementing Mathematics with the NuPRL Proof Development System* (Prentice-Hall, Englewood Cliffs, NJ, 1986).

[15] S. Feferman, Transfinite recursive progressions of axiomatic theories, *J. Symbolic Logic* **27** (1962) 259–316.

[16] A. Felty, Implementing tactics and tacticals in a higher-order logic programming language, *J. Autom. Reasoning* **11** (1993) 43–81.

[17] F. Giunchiglia, The GETFOL Manual - GETFOL version 1, Tech. Rept. 92-0010, DIST, University of Genova, Genoa (1992).

[18] F. Giunchiglia and A. Armando, A conceptual architecture for introspective systems, Tech. Rept., IRST, Trento (1994).

[19] F. Giunchiglia and A. Cimatti, HGKM Manual - a revised version, Tech. Rept. 8906-22, IRST, Trento (1989).

[20] F. Giunchiglia and A. Cimatti, Introspective metatheoretic reasoning, in: *Proceedings META-94, Workshop on Metaprogramming in Logic*, Pisa (1994); also: Tech. Rept. 9211-21, IRST, Trento.

[21] F. Giunchiglia and L. Serafini, Multilanguage hierarchical logics (or: how we can do without modal logics), *Artif. Intell.* **65** (1994) 29–70.

[22] F. Giunchiglia, L. Serafini and A. Simpson, Hierarchical meta-logics: intuitions, proof theory and semantics, in: *Proceedings META-92, Workshop on Metaprogramming in Logic*, Uppsala, Lecture Notes in Computer Science **649** (Springer, Berlin, 1992) 235–249; also: Tech. Rept. 9101-05, IRST, Trento.

[23] F. Giunchiglia and A. Smaill, Reflection in constructive and non-constructive automated reasoning, in: H. Abramson and M.H. Rogers, eds., *Proceedings META-88, Workshop on Metaprogramming in Logic* (MIT Press, Cambridge, MA, 1988) 123–145; also: Tech. Rept. 8902-04, IRST, Trento and DAI Research Paper 375, University of Edinburgh, Edinburgh.

[24] F. Giunchiglia and P. Traverso, Plan formation and execution in a uniform architecture of declarative metatheories, in: M. Bruynooghe, ed., *Proceedings META-90. Workshop on Metaprogramming in Logic* (1990) 306–322; also: Tech. Rept. 9003-12, IRST, Trento.

[25] F. Giunchiglia and P. Traverso, Program tactics and logic tactics, in: *Proceedings 5th International Conference on Logic Programming and Automated Reasoning (LPAR'94)*, Kiev (1994); also: Tech. Rept. 9301-01, IRST, Trento, presented at the Third International Symposium on Artificial Intelligence and Mathematics, Fort Lauderdale, FL (1994).

[26] F. Giunchiglia and R.W. Weyhrauch, A multi-context monotonic axiomatization of inessential non-monotonicity, in: D. Nardi and P. Maes, eds., *Meta-Level Architectures and Reflection* (North-Holland, Amsterdam, 1988) 271–285; also: Tech. Rept. 9105-02, DIST, University of Genova, Genova.

[27] F. Giunchiglia and R.W. Weyhrauch, FOL User Manual - FOL version 2, Manual 9109-08, IRST, Trento (1991); also: Tech. Rept. 91-0006, DIST, University of Genova, Genova.

[28] K. Gödel, Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I, *Monatsh. Math. Phys.* **38** (1931) 173–98 (English translation in [51]).

[29] J. Goguen, Higher-order functions considered unnecessary for higher-order programming, in: D.A. Turner, ed., *Research Topics in Functional Programming* (Addison-Wesley, Reading, MA, 1990) 309–351.

[30] J. Goguen, A. Stevens, H. Hilbrdink and K. Hobley, 2OBJ: a metalogical framework theorem prover based on equational logic, *Phil. Trans. R. Soc. Lond.* **339** (1992) 69–86.

[31] M.J. Gordon, A proof generating system for higher-order logic, in: G. Birtwistle and P.A. Subrahmanyam, eds., *VLSI Specification and Synthesis* (Kluwer, Dordrecht, 1987).

[32] M.J. Gordon, A.J. Milner and C.P. Wadsworth, *Edinburgh LCF - A Mechanized Logic of Computation*, Lecture Notes in Computer Science **78** (Springer, Berlin, 1979).

[33] R. Harper, F. Honsel and G. Plotkin, A framework for defining logics, in: *Symposium on Logic in Computer Science* (1971) 194–204.

[34] R. Harper, D. McQueen and Robin Milner, Standard ML, LFCS report series ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, Edinburgh (1986).

[35] P.M. Hill and J.W. Lloyd, Analysis of meta-programs, in: J. Lloyd, ed., *Proceedings META-88, Workshop on Metaprogramming in Logic* (MIT Press, Cambridge, MA, 1989).

[36] P.M. Hill and J.W. Lloyd, The Gödel programming language, Tech. Rept. CSTR 92-27, Department of Computer Science, University of Bristol, Bristol (1992).

[37] D.J. Howe, Computational metatheory in Nuprl, in: R. Lusk and R. Overbeek, eds., *CADE9* (1988).

[38] S. Jagannathan and G. Agha, A reflective model of inheritance, in: *The Sixth European Conference on Object-Oriented Programming* (1992), Lecture Notes in Computer Science (Springer, Berlin, to appear).

[39] S.C. Kleene, *Introduction to Metamathematics* (North-Holland, Amsterdam, 1952).

[40] T.B. Knoblock and R.L. Constable, Formalized metatheory in type theory, Tech. Rept. TR 86-742, Department of Computer Science, Cornell University, Ithaca, NY (1986).

[41] S.A. Kripke, Outline of a theory of truth, in: *Recent Essays on Truth and the Liar Paradox* (Oxford University Press, Oxford, 1984) 53–82.

[42] Z. Manna, *Mathematical Theory of Computation* (McGraw-Hill, New York, 1974).

[43] L. Paulson, Tactics and tacticals in Cambridge LCF, Tech. Rept. 39, Computer Laboratory, University of Cambridge, Cambridge (1979).

[44] L. Paulson, The foundation of a generic theorem prover, *J. Autom. Reasoning* **5** (1989) 363–396.

[45] L.C. Paulson, A higher-order implementation of rewriting, *Sci. Comput. Program.* **3** (1983) 119–149.

[46] D. Prawitz, *Natural Deduction - A Proof Theoretical Study* (Almquist and Wiksell, Stockholm, 1965).

[47] B.C. Smith, Reflection and semantics in LISP, in: *Proceedings 11th ACM POPL* (1983) 23–35.

[48] M.J. Stefik, Planning and meta-planning, *Artif. Intell.* **16** (1981) 141–169.

[49] C. Talcott, The essence of RUM: theory of the intensional and extensional aspects of LISP-type computation, Ph.D. Thesis, Department of Computer Science, Stanford University, Stanford, CA (1985); also: report STAN-CS-85-1060.

[50] A. Tarski, *Logic, Semantics, Metamathematics* (Oxford University Press, Oxford, 1956).

[51] J. Van Heijenoort, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931* (Harvard University Press, Cambridge, MA, 1967).

[52] L. Viganò, Sintesi ed esecuzione di strategie di prova nella metateoria formale di un dimostratore interattivo, Thesis, University of Genoa, Genoa (1994).

[53] J. von Wright, Representing higher-order logic proofs in HOL, Tech. Rept. jan-18-94, Abo Akademi University, Turku (1994).

[54] R.W. Weyhrauch, Prolegomena to a theory of mechanized formal reasoning, *Artif. Intell.* **13** (1980) 133–176.

[55] R.W. Weyhrauch, An example of FOL using metatheory. Formalizing reasoning and introducing derived inference rules, in: *Proceedings 6th Conference on Automatic Deduction*, New York (1982).

[56] R.W. Weyhrauch and C. Talcott, HGKM: a simple implementation, FOL working paper 4 (1985).

[57] A. Yonezawa, A reflective object oriented concurrent language, *Lecture Notes in Computer Science* **441** (Springer, Berlin, 1991) 254–256.