[22] M.J. Gordon, R. Milner, L. Morris, and C. Wadsworth. A Metalanguage for Interactive Proof in LCF. CSR report series CSR-16-77, Department of Artificial Intelligence, Dept. of Computer Science, University of Edinburgh, 1977.

[23] D. J. Howe. Computational metatheory in Nuprl. In R. Lusk and R. Overbeek, editors, *CADE9*, 1988.

[24] M. Kerber and M. Kohlhase. A Mechanization of Strong Kleene Logic for Partial Functions. In A. Bundy, editor, *Proc. of the 12th Conference on Automated Deduction*, pages 371–385. Springer-Verlag, 1994.

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

[26] E. Melis. A model of analogy-driven proof-plan construction. In *Proc. of the 14th International Joint Conference on Artificial Intelligence*, 1995.

[27] L. Paulson. Tactics and Tacticals in Cambridge LCF. Technical Report 39, Computer Laboratory, University of Cambridge, 1979.

[28] L. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5:363–396, 1989.

[29] L. Paulson. Designing a Theorem Prover. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II, pages 416–475. Oxford University Press, 1992.

[30] D. Prawitz. *Natural Deduction - A proof theoretical study.* Almquist and Wiksell, Stockholm, 1965.

[31] J. von Wright. Representing higher-order logic proofs in HOL. Technical Report jan-18-94, Abo Akademi University, Turku, Finland, 1994.

[32] R.W. Weyhrauch. Prolegomena to a Theory of Mechanized Formal Reasoning. *Artificial Intelligence*, 13(1):133–176, 1980.

[10] A. Felty and D. Howe. Tactic Theorem Proving with Refinement Tree Proofs and Metavariables. In A. Bundy, editor, *Proc. of the 12th Conference on Automated Deduction*, pages 605–619. Springer-Verlag, 1994.

[11] A. Felty and D. Miller. Specifying Theorem Provers in a Higher-Order Logic Programming Language. In R. Luck and R. Overbeek, editors, *Proc. of the 9th Conference on Automated Deduction*, pages 61–80. Springer-Verlag, 1988.

[12] F. Giunchiglia. The `GETFOL` Manual - `GETFOL` version 1. Technical Report 92-0010, DIST - University of Genova, Genoa, Italy, 1992.

[13] F. Giunchiglia and A. Armando. A Conceptual Architecture for Introspective Systems. Forthcoming IRST-Technical Report, 1993.

[14] F. Giunchiglia and A. Cimatti. `HGKM` Manual - a revised version. Technical Report 8906-22, IRST, Trento, Italy, 1989.

[15] F. Giunchiglia and A. Cimatti. Introspective Metatheoretic Reasoning. In *Proc. of META-94, Workshop on Metaprogramming in Logic*, Pisa, Italy, June 19-21, 1994. Also IRST-Technical Report 9211-21, IRST, Trento, Italy.

[16] F. Giunchiglia and P. Traverso. Reflective reasoning with and between a declarative metatheory and the implementation code. In *Proc. of the 12th International Joint Conference on Artificial Intelligence*, pages 111–117, Sydney, 1991. Also IRST-Technical Report 9012-03, IRST, Trento, Italy.

[17] F. Giunchiglia and P. Traverso. A Metatheory of a Mechanized Object Theory. *Artificial Intelligence, to appear*, 1995. IRST-Technical Report 9211-24, IRST, Trento, Italy, 1992.

[18] J. Goguen. Higher-order functions considered unnecessary for higher-order programming. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 309–351. Addison Wesley, 1990.

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

[20] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. Introducing OBJ. In J. Goguen, D. Coleman, and R.Gallimore, editors, *Applications of algebraic specification using OBJ*. Cambridge, 1992.

[21] M.J. Gordon, A.J. Milner, and C.P. Wadsworth. *Edinburgh LCF - A mechanized logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer Verlag, 1979.

2. Π_1 is not a proof. From the induction hypotheses we have that $\vdash_{\mathrm{MT}} \tau_{\pi_1} = fail$. If s_0 is either an axiom or an assumption, then τ_π is $apply(then(t_{\pi_1}, \text{“}t_\rho\text{”}), \text{“}s_0\text{”})$ and $\vdash_{\mathrm{MT}} Tac(\text{“}s_0\text{”})$. If s_0 is neither an axiom nor an assumption, then τ_π is $apply(then(t_{\pi_1}, \text{“}t_\rho\text{”}), fail)$ with $\vdash_{\mathrm{MT}} Tac(fail)$. In both cases, from axiom (A9), the induction hypotheses and by applying ifE we have $\vdash_{\mathrm{MT}} \tau_\pi = fail$

Q.E.D.

References

[1] A. Armando. *Architetture Riflessive per la Deduzione Automatica.* PhD thesis, DIST - University of Genoa, 1993.

[2] D. Basin and R. Constable. Metalogical Frameworks. In *Proceedings of the Second Workshop on Logical Frameworks*, Edinburgh, Scotland, 1991. To Appear as a chapter in a Cambridge University Press book.

[3] R.S. Boyer and J.S. Moore. *A Computational Logic.* Academic Press, 1979. ACM monograph series.

[4] 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, editors, *The correctness problem in computer science*, pages 103–184. Academic Press, 1981.

[5] R.S. Boyer and J.S. Moore. A theorem prover for a computational logic. In *Proceedings of the 10th Conference on Automated Deduction, Lecture Notes in Computer Science 449, Springer-Verlag*, pages 1–15, 1990.

[6] A. Bundy. The Use of Explicit Plans to Guide Inductive Proofs. In R. Luck and R. Overbeek, editors, *Proc. of the 9th Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available as DAI Research Paper No. 349, Dept. of Artificial Intelligence, Edinburgh.

[7] R. Cartwright and J. McCarthy. Recursive Programs as Functions in a First Order Theory, March 1979. SAIL MEMO AIM-324. Also available as CS Dept. Report No. STAN-CS-79-17.

[8] R.L. Constable, S.F. Allen, H.M. Bromley, et al. *Implementing Mathematics with the NuPRL Proof Development System.* Prentice Hall, 1986.

[9] A. Felty. Implementing Tactics and Tacticals in a Higher-Order Logic Programming Language. *Journal of Automated Reasoning*, 11:43–81, 1993.

Theorem C.1 *Let* Π *be a sequent tree of* s*. Let* τ_π *be a sequential tactic application 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 Case:** If Π is s, then it must be either an axiom or an assumption. Then $T(\text{“}s\text{”})$ is an axiom of MT. τ_π is $apply(\text{“}idtac\text{”}, \text{“}s\text{”})$. From axioms (A7) and (A5) we have $\vdash_{\mathrm{MT}} apply(\text{“}idtac\text{”}, \text{“}s\text{”}) = \text{“}s\text{”}$.

Step Case: τ_π is $apply(then(t_{\pi_1}, \text{“}t_\rho\text{”}), \text{“}s_0\text{”})$ and τ_{π_1} is $apply(t_{\pi_1}, \text{“}s_0\text{”})$, since Π and Π_1 are proofs. From the induction hypotheses $\vdash_{\mathrm{MT}} \tau_{\pi_1} = \text{“}s_1\text{”}$ and $\vdash_{\mathrm{MT}} T(\text{“}s_1\text{”})$. From axiom (A9):

$$\vdash_{\mathrm{MT}} \tau_\pi = \begin{array}{l} if\ (\text{“}s_1\text{”} = fail) \\ \quad then\ fail\ else\ apply(\text{“}t_\rho\text{”}, \text{“}s_1\text{”}) \end{array} \tag{9}$$

Since $\vdash_{\mathrm{MT}} \text{“}s_1\text{”} \neq fail$, we apply $If\ E_\neg$ and obtain

$$\vdash_{\mathrm{MT}} \tau_\pi = apply(\text{“}t_\rho\text{”}, \text{“}s_1\text{”})) \tag{10}$$

From axiom (A4) and (A3), we have

$$\vdash_{\mathrm{MT}} \tau_\pi = if\ (\neg Fail(\text{“}s_1\text{”}) \wedge P_\rho(\text{“}s_1\text{”}))\ then\ f_\rho(\text{“}s_1\text{”})\ else\ fail \tag{11}$$

We have $\vdash_{\mathrm{MT}} \neg Fail(\text{“}s_1\text{”})$ from axiom (A2) and $\vdash_{\mathrm{MT}} P_\rho(\text{“}s_1\text{”})$ and $\vdash_{\mathrm{MT}} f_\rho(\text{“}s_1\text{”}) = \text{“}s\text{”}$ (since ρ must be applicable to s_1). Therefore (rule $If\ E$) $\vdash_{\mathrm{MT}} \tau_\pi = \text{“}s\text{”}$. From axiom (A1) we have

$$\vdash_{\mathrm{MT}} (T(\text{“}s_1\text{”}) \wedge P_\rho(\text{“}s_1\text{”})) \supset T(f_\rho(\text{“}s_1\text{”})) \tag{12}$$

and then $\vdash_{\mathrm{MT}} T(f_\rho(\text{“}s_1\text{”}))$. Therefore $\vdash_{\mathrm{MT}} T(\text{“}s\text{”})$. Q.E.D.

Theorem C.2 *Let* Π *be a sequent tree of* s*. Let* τ_π *be a sequential tactic application of* Π*. If* Π *is not a proof, then* $\vdash_{\mathrm{MT}} \tau_\pi = fail$

Proof : **Base Case:** If Π is s, then it is neither an axiom nor an assumption. Then τ_π is $apply(\text{“}idtac\text{”}, fail)$ that is provably equal to $fail$ (axioms (A7) and (A5)).

Step Case: We have two cases.

1. Π_1 is a proof. τ_π is $apply(then(t_{\pi_1}, \text{“}t_\rho\text{”}), \text{“}s_0\text{”})$ and $\vdash_{\mathrm{MT}} T(\text{“}s_0\text{”})$ (since s_0 is either an axiom or an assumption). From theorem C.1 we have that $\vdash_{\mathrm{MT}} \tau_{\pi_1} = \text{“}s_1\text{”}$ and $\vdash_{\mathrm{MT}} T(\text{“}s_1\text{”})$. From axioms (A9), (A4), (A3) and by applying rule $If\ E_\neg$ we obtain (11). $\vdash_{\mathrm{MT}} \neg P_\rho(\text{“}s_1\text{”})$ since ρ must not be applicable to s_1. Then (rule $If\ E_\neg$) $\vdash_{\mathrm{MT}} \tau_\pi = fail$.

B Proof of theorem 3.2

We prove theorem 3.2 by showing that $\mathcal{M}$ is a model of MT (theorem B.1).

Theorem B.1 *$\mathcal{M}$ is a model of MT.*

Proof : We prove that any axiom of MT is true in $\mathcal{M}$. Axioms (A1) are trivially true from the interpretation of T, P_ρ and f_ρ. Axiom (A2) is true since $\mathtt{F}$ is distinct from any sequent of OT. Consider axiom (A3). Let x be assigned to $d \in T_{OT} \cup \{\mathtt{F}\}$. If $d \neq \mathtt{F}$ and $d \in g(P_\rho)$, then both $t_\rho(x)$ and the conditional term are interpreted into $\rho_p(d)$. If $d = \mathtt{F}$ or $d \notin g(P_\rho)$, then both $t_\rho(x)$ and the conditional term are interpreted into $\mathtt{F}$. Consider now axiom (A4). Let x be assigned to $d \in T_{OT} \cup \{\mathtt{F}\}$. $apply(\text{“}t_\rho\text{”}, x)$ is interpreted into $F_\rho(d)$. The interpretation of $t_\rho(x)$ is exactly $F_\rho(d)$. Therefore (A4) is true in $\mathcal{M}$. The proof is analogous for (A5) and (A6).

Then we prove that the axioms about tacticals are true in $\mathcal{M}$. Let x, t_i and t_j be assigned to $d \in T_{OT} \cup \{\ \mathtt{F}\ \}$, $d_i \in \mathcal{D}_f$ and $d_j \in \mathcal{D}_f$, respectively. The proof for axioms (A7), (A8) is trivial. Consider axiom (A9). $then(t_i, t_j)$ is interpreted into $F_{then}[d_i, d_j] \in \mathcal{D}_f$. $apply(then(t_i, t_j), x)$ is interpreted into

$$g(apply)(F_{then}[d_i, d_j], d) = \begin{cases} d_j(d_i(d)) & \text{if } d_i(d) \neq \mathtt{F} \\ \mathtt{F} & \text{if } d_i(d) = \mathtt{F} \end{cases}$$

If $d_i(d) \neq \mathtt{F}$, the conditional term is interpreted into $d_j(d_i(d))$. If $d_i(d) = \mathtt{F}$, it is interpreted into $\mathtt{F}$.

The proof for axioms (A10), (A11), (A12) is similar to the proof for axiom (A9).

We prove now that (A13) is true in $\mathcal{M}$. $repeat(t_i)$ is interpreted into $F_{repeat}[d_i] \in \mathcal{D}_f$. $apply(repeat(t_i), x)$ is interpreted into $F_{repeat}[d_i](d)$. But the right hand of the equality in (A13) is interpreted into $F_{repeat}[d_i](d)$ as well. Indeed, if $d_i(d) \neq \mathtt{F}$, then the interpretation of the conditional term is the interpretation of $apply(repeat(t_i), apply(t_i, x))$, i.e. $F_{repeat}[d_i](d_i(d))$; but $F_{repeat}[d_i](d_i(d)) = \Phi_{d_i}(F_{repeat}[d_i])(d) = F_{repeat}[d_i](d)$. If $d_i(d) = \mathtt{F}$, then its interpretation is d and $F_{repeat}[d_i](d) = d$. Axioms (A14)-(A21) are trivially true from the definition of $\mathcal{D}_f$. Q.E.D.

C Proof of theorem 4.1

Theorem C.1 proves parts $(1)(a) \Leftarrow$ and $(1)(b) \Rightarrow$, theorem C.2 corresponds to part $(2)(a) \Leftarrow$. In the proofs, we call Π the sequent tree of s built by applying an inference rule ρ to the sequent s_1 end sequent of Π_1. We call the leaf of Π_1, s_0. We call t_{π_1}, τ_{π_1}, t_π and τ_π the sequential tactics and the sequential tactic applications of Π_1 and Π, respectively. $(2)(b) \Rightarrow$ is a trivial corollary of theorem C.2. $(1)(a) \Rightarrow$, $(1)(b) \Leftarrow$, $(2)(a) \Rightarrow$ and $(2)(b) \Leftarrow$ are trivial corollaries of theorems C.1 and C.2 and theorem 3.2 (proofs by contradiction).

started to develop the family of rewriting functions implemented in Cambridge LCF and described in [29]. A major future goal 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 [4, 23]). 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 [32] (the same theorem is also stated and proved in [2]).

Acknowledgments

The authors thank the Mechanized Reasoning Groups in Trento and Genoa. Members of these groups are working on related issues. Massimo Benerecetti has mechanized MT (extended to allow the use of tacticals) in GETFOL. We thank also Alessandro Armando, Alessandro Cimatti, David Basin, Alan Bundy, Luciano Serafini, Alan Smaill, Carolyn Talcott, Toby Walsh and Richard Weyhrauch for feedback on various aspects of the work described in this paper.

Appendix:

A Proof of theorem 3.1

Proof : We prove that Φ_d is a monotonic function acting on the partial ordering $\sqsubseteq$, i.e. $\varphi_1 \sqsubseteq \varphi_2$ implies $\Phi_d(\varphi_1) \sqsubseteq \Phi_d(\varphi_2)$, where $\varphi_1, \varphi_2 \in \mathcal{D}_f$. We have that

$$\Phi_d(\varphi_1)(x) = \begin{cases} \varphi_1(d(x)) & \text{if } d(x) \neq \mathtt{F} \text{ and } x \neq \mathtt{E} \\ x & \text{if } d(x) = \mathtt{F} \text{ and } x \neq \mathtt{E} \\ \mathtt{E} & \text{if } x = \mathtt{E} \end{cases}$$

and

$$\Phi_d(\varphi_2)(x) = \begin{cases} \varphi_2(d(x)) & \text{if } d(x) \neq \mathtt{F} \text{ and } x \neq \mathtt{E} \\ x & \text{if } d(x) = \mathtt{F} \text{ and } x \neq \mathtt{E} \\ \mathtt{E} & \text{if } x = \mathtt{E} \end{cases}$$

If $d(x) \neq \mathtt{F}$ and $x \neq \mathtt{E}$, then $\Phi_d(\varphi_1)(x) = \varphi_1(d(x)) \sqsubseteq \varphi_2(d(x)) = \Phi_d(\varphi_2)(x)$. If $d(x) = \mathtt{F}$ and $x \neq \mathtt{E}$, or $x = \mathtt{E}$, then $\Phi_d(\varphi_1)(x) = x \sqsubseteq x = \Phi_d(\varphi_2)(x)$. Therefore $\Phi_d(\varphi_1) \sqsubseteq \Phi_d(\varphi_2)$.

Q.E.D.

equational logic. Like MT, 2OBJ supports a first order treatment of tactics. In 2OBJ, tactics are programmed in its equational logic. However, there is no relation between tactics in 2OBJ and the implementation of OBJ3. Moreover the metatheory of 2OBJ has no explicit notion of failure.

From a technical point of view, MT has some features which make it somewhat unusual. Some of these features are the following: Inference rules are functions and not predicates, as it happens for instance in [31]; inference rules do not take theories and signatures as arguments, as it happens for instance in [31] (this in `GETFOL` is solved using the multicontext machinery [12]); even if MT can reason about proofs this notion is not explicitly axiomatized, as it happens for instance in [31, 2]. Finally, the notion of failure (`F`) is explicit, and kept distinct from the notion of partialness (`E`). This is not what happens in most of the approaches which deal with partialness, e.g. [7, 25] (see also [24]).

As stated in the introduction, our ultimate goal (still far from being achieved) is to prove the correctness of the theorem prover within the theorem prover itself. We share this goal with the work in progress on Acl2 [5]. There are various differences between the two approaches. One difference, which is relevant to this paper, is that in Acl2 the logic language and the implementation language are the same. Keeping Logic Tactics and Program Tactics distinct, as we do, seems to provide some advantages, for instance for what concerns how to deal with state. For example, it is possible for us to store in a global variable the set of proven theorems and to reason declaratively about it. The idea is to see state (e.g. the current proof) as storing partial computations relative to a function or predicate (e.g. the provability predicate T). This allows us to lift the code that updates and reads state into axioms which formalize the function and predicate whose computations are approximately represented by the state itself. Some of the details about this issue are in [15, 13]. Another advantage is that we may have Program Tactics which are not translated into Logic Tactics or vice versa. This is a necessary feature in the presence of non-terminating Program Tactics (see the motivations in the introduction).

7 Conclusion and future work

We have described a first order metatheory, called MT, which is expressive enough to represent tactics and tacticals. Tactics are terms of MT (called Logic Tactics) and tacticals are function symbols of MT. MT is expressive enough to represent a proof strategy which does not correspond to a finite composition of proof steps. Moreover, MT can express soundly tactics which do not terminate. MT has been constructed so that it is possible to define a relation between Logic Tactics and `GETFOL` Program Tactics.

At the moment we are studying some more general sufficient conditions for a characterization of recursive (possibly "non terminating") Logic Tactics which preserve consistency. We are also studying the possibility for MT to construct powerful proof strategies as Logic Tactics. We have

tacticals and, in particular, can soundly express strategies based on the recursive application of tactics (through the tactical *repeat*). This has been achieved by several technical extensions to the metatheory presented in [17]. In order to express tacticals in a first order setting, we have extended the logic with *names of tactics* and the function symbol *apply* (see for instance axiom (A4)) which has allowed us to axiomatize Logic Tacticals (see axioms (A7)–(A13)) and Logic Tactics (see axioms (A14)–(A21)). As a consequence, the model of MT has been extended too. Indeed, in [17] the domain of the model is simply the union of the set of terms, wffs and sequents ($\mathcal{D}_0$) with the elements to handle partialness and failure (`E` and `F`). In order to interpret tacticals, we have extended the domain with $\mathcal{D}_f$ (see definition 3.1) and the model with functions defined over $\mathcal{D}_f$ (see definition 3.2). Notice that, the model has been extended in a way to allow for a possible extension of the results presented in [17], keeping and extending the correspondence between Logic and Program Tactics. Moreover, in this paper we have dealt with the problem of (possibly non terminating) recursive applications of tactics by proving that the interpretation of *repeat* is the least fixed point over the partial ordering $\sqsubseteq$ (see theorem 3.1). As an example of the significance of this extension, notice that all the proofs of the form (3) and (4), which have been synthetically expressed by the only Logic Tactic (2), should have to be expressed in [17] with one tactic for each possible resulting sequent tree schema (i.e. in this case an infinite number of tactics). This is an example of how this work makes it feasible in practice to express tactics in a logical first order metatheory.

Program Tactics have been mostly implemented in ML [22] and used successfully in several theorem provers, like `LCF` [21, 27], Isabelle [28] and NuPRL [8]. Logic Tactics have been encoded in higher order logical theories and logic programming languages. In [23], the higher order NuPRL type theory is used as a language for constructing theorem proving procedures. In [11, 9], tactics and tacticals are implemented in λProlog, an extended higher-order logic programming language. In [10], proof trees are represented in a logical theory where "justifications" of proof steps can be representations of tactics. There are two main differences with our work. First, MT is first order. Reasoning about tactics can be done entirely in first order logic. We do not discuss here the advantages of working in a first order setting, see for instance [18, 7]. Second, neither [23] nor [11, 9, 10] provide a relation between Logic Tactics and programs that implement the theorem prover. NuPRL and λProlog cannot reason about and extend/modify their own system code. Beside these main facts, there are also technical (but important) differences between Logic Tactics in MT and tactics in λProlog. Tactics in λProlog specify proof search strategies by providing an interpreter on top of λProlog which is itself interpreted under the fixed λProlog search strategy. As a consequence of this and of the fact that tactics are relational, failure is treated as falsity (the tactical `orelse` is defined as disjunction) and failure handling is performed by backchaining using a depth first search paradigm. Similarly, `then` is defined as conjunction. In MT, the axiomatization of *then* and *orelse* is fully declarative and independent of any system underlying search strategy.

Some work closely related to ours (but which seems at an earlier stage) is the work on 2OBJ [19]. 2OBJ is a tactic-based theorem prover built upon OBJ3 [20], a term rewriting implementation of

$$\forall x\ t(x) = \begin{array}{l} if\ (t(x) = apply(\text{“}idtac\text{”}, x)) \\ then\ apply(\text{“}failtac\text{”}, x) \\ else\ apply(\text{“}idtac\text{”}, x) \end{array} \quad (7)$$

is provable, then MT is inconsistent. Indeed, for any sequent s, from (7), under the assumption $t(\text{“}s\text{”}) = \text{“}s\text{”}$, we prove $\text{“}s\text{”} = fail$ and therefore $\bot$, the sentential constant for falsity. From (7) and the assumption $\neg t(\text{“}s\text{”}) = \text{“}s\text{”}$, we prove $t(\text{“}s\text{”}) = \text{“}s\text{”}$ and therefore $\bot$. Hence from $t(\text{“}s\text{”}) = \text{“}s\text{”} \vee \neg t(\text{“}s\text{”}) = \text{“}s\text{”}$ we prove $\bot$. Notice that the wff above defines a recursive tactic t which corresponds to a Program Tactic that does not terminate. Intuitively, it states that if the tactic t succeeds, then it fails, and if it fails, it succeeds.

Our goal is therefore to provide some sufficient condition for a characterization of recursive (possibly “non terminating”) Logic Tactics which preserve consistency. For instance, we have extended MT with axioms which correspond to a particular class of recursive definitions of Program Tactics:

$$\forall x\ Tac(x) \supset f^*_\rho(x) = \begin{array}{l} if\ P_\rho(x) \\ then\ f^*_\rho(f_\rho(x)) \\ else\ x \end{array} \quad (8)$$

Intuitively, any f^*_ρ corresponds to a recursive application of the tactic which applies ρ until the conditions of applicability (P_ρ) are not satisfied. For instance, consider the following recursive definition of $fandeltac^*$.

$$\forall x\ Tac(x) \supset fandeltac^*(x) = \begin{array}{l} if\ Conj(x) \\ then\ fandeltac^*(fandel(x)) \\ else\ x \end{array}$$

$fandeltac^*$ recursively applies the left conjunction elimination rule until it fails. We can show that $fandeltac^*(x)$ is equivalent to $apply(repeat(\text{“}fandeltac\text{”}), x)$.

The proof that recursive definitions of the form (8) preserve the consistency of MT is trivial. Notice however that (8) captures a still very limited set of possible recursive definitions. At the moment we are studying some more general sufficient conditions for a characterization of recursive (possibly “non terminating”) Logic Tactics which preserve consistency.

6 Related work

This work builds on the results presented in [17]. In this paper we have extended significantly the set of tactics which can be expressed in MT. MT can now express tactics composed through

4. If x is an individual variable of MT and $\tau_1, \tau_2 \in \mathcal{TA}$, then $let\ \ x = \tau_1\ \ in\ \ \tau_2 \in \mathcal{TA}$.

$\mathcal{TA}$ includes logic tactic applications as defined in section 4 (item 1. in the definition above), it allows for terms with variables (item 2.), conditionals (item 3.) and an environment term constructor *let* (item 4.). The elimination and introduction rules for conditionals have been given in section 2, figure 1. In figure 3, we extend $\mathcal{MR}$ with inference rules to eliminate and introduce *let*. The correctness of these rules is proved in [1]. Notice that *let* can be used to axiomatize tacticals. For

$$\frac{P(t_2[t_1])}{P(let\ \ x = t_1\ \ in\ \ t_2[x])}\ let\ I \qquad \frac{P(let\ \ x = t_1\ \ in\ \ t_2[x])}{P(t_2[t_1])}\ let\ E$$

with the restriction on the let I rule that x does not appear free in t_2.

Figure 3: *let* inference rules

instance, $(A13)$ can be replaced with the following axiom:

$$\begin{aligned}\forall x \forall y \forall t_i\ (Tac(x) \wedge LTac(t_i) \supset \\ apply(repeat(t_i), x) = let\ \ y = apply(t_i, x)\ \ in \\ if\ (y = fail) \\ then\ x \\ else\ apply(repeat(t_i), y))\end{aligned}$$

As a further example, consider the following axiom:

$$\begin{aligned}\forall x \forall y \forall t_i \forall t_j\ (Tac(x) \wedge LTac(t_i) \wedge LTac(t_j) \supset \\ apply(!(t_i, t_j), x) = let\ \ y = apply(t_i, x)\ \ in \\ if\ (y = fail) \\ then\ apply(!(t_i, t_j), apply(t_j, x)) \\ else\ y)\end{aligned}$$

The construct ! applies iteratively t_i and t_j till t_i succeeds. It has a similar behaviour to the ML construct `!` [22]. It allows for failure trap with re-iteration.

Second, we consider the problem of recursive definitions. The tactical *repeat* is not the only way to construct repeated applications of tactics. Program Tactics can also be defined recursively. On the other hand, in general, allowing for recursive definitions of Logic Tactics and logic tactic applications in MT may not preserve consistency. For instance, if the following wff

Corollary 4.1 (failure and success for logic tactic applications) *Let* Π *be a sequent tree of* s. *Let* τ_π *be the sequential tactic application of* Π. *Let* τ *be a tactic application. If* $\vdash_{\mathrm{MT}} \tau = \tau_\pi$, *then*

$$\begin{array}{lcccl} \vdash_{\mathrm{MT}} \tau = \text{“}s\text{”} & \Longleftrightarrow & \Pi \textit{ is a proof of } s. & \Longleftrightarrow & \vdash_{\mathrm{MT}} T(\tau) \\ \vdash_{\mathrm{MT}} \tau = fail & \Longleftrightarrow & \Pi \textit{ is not a proof.} & \Longleftrightarrow & \vdash_{\mathrm{MT}} \neg T(\tau) \end{array}$$

A Program Tactic succeeds iff it builds a sequent tree (Π) which is a proof of a theorem (s). A Logic Tactic application (τ) is provably equal to the constant denoting the theorem (“s”) iff it corresponds to a sequent tree Π which is a proof of s. A Program Tactic fails when it tries to apply some inference rule that is not applicable, i.e. when it tries to build a sequent tree Π that is not a proof. A Logic Tactic application is provably equal to failure ($fail$) iff it corresponds to a sequent tree Π which is not a proof.

Notice that, under the hypotheses of corollary 4.1, we can prove that $\vdash_{\mathrm{MT}} T(\tau) \vee Fail(\tau)$, i.e. $\vdash_{\mathrm{MT}} Tac(\tau)$, that is that τ either succeeds or fails. This is actually what happens with Program Tactics that terminate. However, this may not be the case. Consider, for instance, the Logic Tactic $repeat($“$idtac$”$)$. Since “$idtac$” applied to a given sequent always succeeds, the corresponding Program Tactic applies “$idtac$” an infinite number of times. In other words, the sequential tactic application $apply(repeat($“$idtac$”$), $“$s$”$)$ does not correspond to any sequent tree. Any sequential tactic application τ_π corresponds to a finite number of applications of inference rules. Therefore, a τ_π such that $\vdash_{\mathrm{MT}} apply(repeat($“$idtac$”$), $“$s$”$) = \tau_\pi$ does not exist and corollary 4.1 cannot be applied. The condition $\vdash_{\mathrm{MT}} \tau = \tau_\pi$ of corollary 4.1 captures the fact that the Program Tactic corresponding to τ terminates. A lot of work has been done on providing conditions for and on proving the termination of recursive programs (e.g. see [3]). Our approach is different. We allow for Logic Tactics which correspond to Program Tactics which are not guaranteed to terminate. Then, when needed and when possible, we prove that a logic tactic application corresponds to an application of a Program Tactic which terminates ($\vdash_{\mathrm{MT}} \tau = \tau_\pi$).

5 Some extensions

Tacticals constitute a powerful and well-tested mechanism for composing Program Tactics. Nevertheless, Program Tactics are written using also traditional programming language constructs, e.g. conditionals, environment constructors, loops and recursive definitions. We want the same ability in MT. First, we extend logic tactic applications. We call the extended set, $\mathcal{TA}$.

1. If $t \in \mathcal{T}$ and s is a sequent of OT, then $apply(t, \text{“}s\text{”}) \in \mathcal{TA}$.
2. If $t \in \mathcal{T}$ and x is an individual variable of MT, then $apply(t, x) \in \mathcal{TA}$.
3. If A is a wff of MT and $\tau_1, \tau_2 \in \mathcal{TA}$, then *if* A *then* τ_1 *else* $\tau_2 \in \mathcal{TA}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\rightarrow(...(A_1(x_1)\wedge ...)\wedge A_{n-2}(x_{n-2}))\wedge A_{n-1}(x_{n-1}))\wedge A_n(x_n)}{\rightarrow(...(A_1(x_1)\wedge ...)\wedge A_{n-2}(x_{n-2}))\wedge A_{n-1}(x_{n-1})}\,\forall E}{\vdots}\,\forall E}{\rightarrow A_1(x_1)\wedge A_2(x_2)}\,\forall E}{\rightarrow A_1(x_1)}\,\forall E \qquad (6)$$

We associate to every object level sequent tree Π a *sequential (logic) tactic* t_π in MT. Sequential logic tactics of object level sequent trees are defined inductively over the structure of sequent trees. In the base case, a sequent tree is a single sequent. If the sequent is either an axiom or an assumption, then the sequent tree is a proof. If it is neither an axiom nor an assumption, then the sequent tree is not a proof. In both cases, its sequential tactic is "$idtac$". In the step case, if t_{π_1} is the sequential tactic of Π_1, and Π is built from Π_1 by applying ρ to the end sequent of Π_1, then $then(t_{\pi_1}, \text{“}t_\rho\text{”})$ is the sequential tactic of Π. For instance, if a sequent tree is built by applying first the rule ρ_1 to an axiom or an assumption, and then ρ_2 and ρ_3 are applied in the sequent tree in the given order, then the corresponding sequential tactic is $then(then(then(\text{“}idtac\text{”}, \text{“}t_{\rho_1}\text{”}), \text{“}t_{\rho_2}\text{”}), \text{“}t_{\rho_3}\text{”})$.

Program Tactics, when executed, are applied to given arguments. Similarly, in MT, we define a *sequential (logic) tactic application* τ_π of Π. Let s be the leaf of Π. Let t_π is the sequential tactic of Π. τ_π is $apply(t_\pi, \text{“}s\text{”})$, if s is an axiom or an assumption. τ_π is $apply(t_\pi, fail)$, if s is neither an axiom nor an assumption. A *(logic) tactic application* τ is a term of the form $apply(t, \text{“}s\text{”})$, where $t \in \mathcal{T}$ and s is a sequent. We say that Π is a sequent tree of s if s is the end sequent of Π. We prove that sequential tactic applications have the right behaviour.

Theorem 4.1 (failure and success for sequential logic tactic applications) *Let* Π *be a sequent tree of* s. *Let* τ_π *be the sequential tactic application of* Π. *Then*

$$\begin{array}{lllllll}
(1) & \vdash_{\mathrm{MT}} \tau_\pi = \text{“}s\text{”} & \Longleftrightarrow_{(a)} & \Pi \textit{ is a proof of } s. & \Longleftrightarrow_{(b)} & \vdash_{\mathrm{MT}} T(\tau_\pi) \\
(2) & \vdash_{\mathrm{MT}} \tau_\pi = fail & \Longleftrightarrow_{(a)} & \Pi \textit{ is not a proof.} & \Longleftrightarrow_{(b)} & \vdash_{\mathrm{MT}} \neg T(\tau_\pi)
\end{array}$$

Part (1) of theorem 4.1 states that a successful tactic corresponding to a proof can be proved equal to the name of a sequent (part (1)(a)) that is a theorem of OT (part (1)(b)). ($\vdash_{\mathrm{MT}} T(\text{“}s\text{”}) \Longleftrightarrow \vdash_{\mathrm{OT}} s$, where $\vdash_{\mathrm{OT}}$ stands for provability in OT, is a corollary of theorems 3.2 and 4.1). Part (2) states that a tactic that does not correspond to a proof can be proved equal to failure (part (2)(a)) and it does not denote an object level theorem (part (2)(b)). Notice that the fact that a Program Tactic fails to prove a sequent does not imply that the sequent is not provable. Analogously, part (2) states that the tactic does not denote a theorem ($\neg T(\tau_\pi)$), but *does not state* the much stronger fact that s is not a theorem ($\neg T(\text{“}s\text{”})$). Theorem 4.1 can be generalized to logic tactic applications.

the standard approaches and the approach undertaken here is that we have two distinct special elements, `E` and `F`. From a theoretical point of view, we could construct a metatheory and a model where `E` and `F` are collapsed in a unique element. Nevertheless, the distinction between `E` and `F` is important in order to define a correspondence between Logic and Program Tactics. `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 must be partial. On the contrary, `F` is denoted by $fail$ in MT and is implemented by a data structure in the `GETFOL` code. This data structure is a witness of observable failures. It is returned by and passed as argument to Program Tactics. F_F, F_ρ, $F_{then}[d_1, d_2]$, $F_{orelse}[d_1, d_2]$, $F_{repeat}[d_1]$, $F_{try}[d_1]$ and $F_{progress}[d_1]$ specify how Program Tactics and tacticals must deal with this data structure.

Logic Tacticals are assigned to functions (e.g. $g(then)$, $g(orelse)$ and $g(repeat)$) that, given elements of D_f (denoted by Logic Tactics), return an element of the same set D_f. Therefore, intuitively, tacticals are interpreted according to their original intended meaning, i.e. as functions that given tactics return tactics. Moreover, notice that elements of $\mathcal{D}_f \subseteq \mathcal{D}$ are functions over $(T_{OT} \cup \{\mathtt{F}\}) \subseteq \mathcal{D}$. $g(apply)$ allows us to apply these functions to their arguments. This fact, plus the fact that Logic Tactics denote elements of $\mathcal{D}_f$, allows us to interpret Logic Tactics and Tacticals in a first order setting.

4 Success and failure of logic tactic applications

In this section we prove some general properties of Logic Tactics. First we provide a formal notion of "logic tactic application", i.e. the logical analogous of applications of Program Tactics to some arguments. Then we prove the following facts: a logic tactic application is provably equal to the name of a theorem of OT iff it corresponds to a proof of the theorem in OT; it is provably equal to failure iff it corresponds to a tree of rule applications that is not a proof; (non) termination of logic tactic applications is captured formally in MT by the (un)provability of certain statements.

Program Tactics, when executed, build trees of inference rule applications where either all the rules are applicable (the tactic succeeds) or there is a rule which is not applicable (the tactic fails). If all the rules are applicable, the sequent tree is a proof. If there exists a rule which is not applicable, the sequent tree is *not* a proof. A *sequent tree* Π is defined formally as a tree of sequents, each labeled by an inference rule. OT proofs (3) adn (4) are examples of sequent trees. An example of a sequent tree which is not a proof is the following:

$$g(repeat)(d) = \begin{cases} F_{repeat}[d] & if \quad d \in \mathcal{D}_f \\ \mathtt{E} & otherwise \end{cases}$$

16. $g(try)$ is a function over $\mathcal{D}$ such that

$$g(try)(d) = \begin{cases} F_{try}[d] & if \quad d \in \mathcal{D}_f \\ \mathtt{E} & otherwise \end{cases}$$

17. $g(progress)$ is a function over $\mathcal{D}$ such that

$$g(progress)(d) = \begin{cases} F_{progress}[d] & if \quad d \in \mathcal{D}_f \\ \mathtt{E} & otherwise \end{cases}$$

18. $g(apply)$ is a function over $\mathcal{D} \times \mathcal{D}$ such that

$$g(apply)(d_1, d_2) = \begin{cases} d_1(d_2) & if \quad d_1 \in \mathcal{D}_f \text{ and } d_2 \in T_{OT} \cup \{\mathtt{F}\} \\ \mathtt{E} & otherwise \end{cases}$$

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

Theorem 3.2 *MT is consistent.*

We prove the consistency of MT by showing that $\mathcal{M} =< \mathcal{D}, g >$ is a model of MT.

$\mathcal{M}$ has been built to identify some requirements that the code implementing Program Tactics must satisfy in order to allow for a relation with Logic Tactics. `E` has been introduced to handle non termination and totalize function symbols (e.g. f_ρ) that otherwise would be partially defined. f_ρ corresponds to a function in the `GETFOL` code that is undefined for some inputs. For instance, universal specialization is implemented by means of a function that, given a (data structure corresponding to) a universally quantified wff, returns (a data structure corresponding to) its matrix. This function is partial since it is undefined for (data structures corresponding to) wffs that are not universally quantified. Partialness is a general characteristic of a large amount of the code of `GETFOL` (and of any running system). For instance, this is the case also for tacticals, which are defined only over tactics, and for *apply*, which is defined only over pairs of tactics and theorems or failures. Partialness allows us to achieve efficiency (the code does not have to test and decide for all the possible inputs). Extending the domain with `E` to handle partial functions and non termination is a well known standard technique (see, for instance, [7, 25]). One essential difference between

5. *$g(=)$ is the identity relation over $\mathcal{D}$.*

6. *$g(LTac) = \mathcal{D}_f$.*

7. *$g(P_\rho)$ is the subset of sequents ρ is applicable to.*

8. *$g(f_\rho)$ is the function over $\mathcal{D}$ that returns the conclusion of ρ, if ρ is applicable to $d \in \mathcal{D}$. It returns* `E`*, otherwise. Let ρ_p be the partial function, defined only over $g(P_\rho)$, that returns the conclusion of ρ. Then*

$$g(f_\rho)(d) = \begin{cases} \rho_p(d) & \text{if} \quad d \in g(P_\rho) \\ \texttt{E} & \text{otherwise} \end{cases}$$

9. *$g(t_\rho)$ is the function over $\mathcal{D}$ such that,*

$$g(t_\rho)(d) = \begin{cases} \rho_p(d) & \text{if } d \in T_{OT} \cap g(P_\rho) \\ \texttt{F} & \text{if } d \in (T_{OT} \perp g(P_\rho)) \cup \{\texttt{F}\} \\ \texttt{E} & \text{otherwise} \end{cases}$$

10. *$g(idtac)$ is the function over $\mathcal{D}$ such that*

$$g(idtac)(d) = \begin{cases} d & \text{if } d \in T_{OT} \cup \{\ \texttt{F}\ \} \\ \texttt{E} & \text{otherwise} \end{cases}$$

11. *$g(failtac)$ is the function over $\mathcal{D}$ such that*

$$g(failtac)(d) = \begin{cases} \texttt{F} & \text{if } d \in T_{OT} \cup \{\ \texttt{F}\ \} \\ \texttt{E} & \text{otherwise} \end{cases}$$

12. *$g(\text{“}idtac\text{”}) = F_{id}$, $g(\text{“}failtac\text{”}) = F_F$ and $g(\text{“}t_\rho\text{”}) = F_\rho$.*

13. *$g(then)$ is a function over $\mathcal{D} \times \mathcal{D}$ such that*

$$g(then)(d_1, d_2) = \begin{cases} F_{then}[d_1, d_2] & \text{if} \quad d_1, d_2 \in \mathcal{D}_f \\ \texttt{E} & \text{otherwise} \end{cases}$$

14. *$g(orelse)$ is a function over $\mathcal{D} \times \mathcal{D}$ such that*

$$g(orelse)(d_1, d_2) = \begin{cases} F_{orelse}[d_1, d_2] & \text{if} \quad d_1, d_2 \in \mathcal{D}_f \\ \texttt{E} & \text{otherwise} \end{cases}$$

15. *$g(repeat)$ is a function over $\mathcal{D}$ such that*

7. $F_{progress}[d](x) = \begin{cases} d(x) & \text{if } d(x) \neq x \text{ and } x \neq \mathtt{E} \\ \mathtt{F} & \text{if } d(x) = x \text{ and } x \neq \mathtt{E} \\ \mathtt{E} & \text{if } x = \mathtt{E} \end{cases}$

8. $F_{repeat}[d](x) = f_{\Phi_d}$, *where* f_{Φ_d} *is the least fixpoint of the functional* Φ_d *over* $[T_{OT} \cup \{\mathtt{F}\} \cup \{\mathtt{E}\} \rightarrow T_{OT} \cup \{\mathtt{F}\} \cup \{\mathtt{E}\}]$, *defined as*

$$\Phi_d(\varphi)(x) = \begin{cases} \varphi(d(x)) & \text{if } d(x) \neq \mathtt{F} \text{ and } x \neq \mathtt{E} \\ x & \text{if } d(x) = \mathtt{F} \\ \mathtt{E} & \text{if } x = \mathtt{E} \end{cases}$$

with φ *being a function variable, and the partial ordering* $\sqsubseteq$ *on* $T_{OT} \cup \{\mathtt{F}\} \cup \{\mathtt{E}\}$ *is defined as*

$$\mathtt{E} \sqsubseteq d \text{ and } d \sqsubseteq d \text{ for all } d \in T_{OT} \cup \{\mathtt{F}\} \cup \{\mathtt{E}\}$$

Some explanations are in order. We use the notation $f[\ ]$ to denote elements of $\mathcal{D}_f$. For each $d_1, d_2 \in \mathcal{D}_f$, we have an element $F_{then}[d_1, d_2]$, $F_{orelse}[d_1, d_2]$, $F_{repeat}[d_1]$, $F_{try}[d_1]$ and $F_{progress}[d_1]$ in $\mathcal{D}_f$. We use the notation $f(\ )$ to denote function application. If $d_1, d_2 \in \mathcal{D}_f$ and $x \in T_{OT} \cup \{\mathtt{F}\} \cup \{\mathtt{E}\}$, then $d_1(x)$ stands for "the function d_1 applied to the argument x", and $d_2(d_1(x))$ stands for the composition $d_2 \circ d_1$ applied to x.

The following theorem proves that Φ_d is monotonic, and therefore has a least fixpoint.

Theorem 3.1 Φ_d *is monotonic over the partial ordering* $\sqsubseteq$ *on* $T_{OT} \cup \{\mathtt{F}\} \cup \{\mathtt{E}\}$ *defined as*

$$\mathtt{E} \sqsubseteq d \text{ and } d \sqsubseteq d \text{ for all } d \in T_{OT} \cup \{\mathtt{F}\} \cup \{\mathtt{E}\}$$

The next step is to define the interpretation function g.

Definition 3.2 (Interpretation function g of $\mathcal{M}$)

1. $g(\text{“}s\text{”}) = s$, $g(\text{“}w\text{”}) = w$ *and* $g(\text{“}t\text{”}) = t$, *where* s, w *and* t *are any sequent, wff and term of OT, respectively.*

2. $g(fail) = \mathtt{F}$.

3. $g(Seq)$ *is the set of sequents of OT.*

4. $g(T) = T_{OT}$, *where* T_{OT} *is the set of theorems of OT.*

The domain of interpretation $\mathcal{D}$ includes a set (called $\mathcal{D}_o$) of objects of the object theory OT, e.g. the sequent $\Gamma \longrightarrow \forall x A(x)$. These objects are denoted by terms of MT, e.g. the constant "$\Gamma \longrightarrow \forall x A(x)$". The domain contains the two special elements $\mathtt{E}$ and $\mathtt{F}$. $\mathtt{E}$ intuitively means "undefined" and is used to handle partialness and non termination. $\mathtt{F}$ is used to interpret failure, i.e. the constant $fail$ of MT. The domain includes a further subset, called $\mathcal{D}_f$, which is used to interpret Logic Tactics. Tacticals are interpreted as functions defined over elements of $\mathcal{D}_f$.

Definition 3.1 (Domain $\mathcal{D}$ of $\mathcal{M}$) *$\mathcal{D} = \mathcal{D}_o \cup \{\mathtt{E}\} \cup \{\mathtt{F}\} \cup \mathcal{D}_f$. $\mathcal{D}_o$ is the set of terms, wffs and sequents of OT. $\mathtt{E}$ and $\mathtt{F}$ are distinct from any other element of $\mathcal{D}$.*

$\mathcal{D}_f$ is constructed inductively as follows.

1. $F_{id} \in \mathcal{D}_f, \quad F_F \in \mathcal{D}_f, \quad \{F_\rho\} \subseteq \mathcal{D}_f$.
2. *If d_1 and $d_2 \in \mathcal{D}_f$, then $F_{then}[d_1, d_2] \in \mathcal{D}_f$, $F_{orelse}[d_1, d_2] \in \mathcal{D}_f$, $F_{repeat}[d_1] \in \mathcal{D}_f$, $F_{try}[d_1] \in \mathcal{D}_f$ and $F_{progress}[d_1] \in \mathcal{D}_f$.*

where we have an element F_ρ for each $\rho \in \mathcal{R}$. The members of $\mathcal{D}_f$ are functions over the set of theorems of OT union the element $\mathtt{F}$ union $\mathtt{E}$, i.e. functions from $T_{OT} \cup \{\mathtt{F}\} \cup \{\mathtt{E}\}$ to $T_{OT} \cup \{\mathtt{F}\} \cup \{\mathtt{E}\}$, where T_{OT} is the set of theorems of OT. They are defined as follows.

Let $x \in T_{OT} \cup \{\mathtt{F}\} \cup \{\mathtt{E}\}$:

1. $F_{id}(x) = x$
2. $F_F(x) = \begin{cases} \mathtt{F} & \text{if } x \neq \mathtt{E} \\ \mathtt{E} & \text{if } x = \mathtt{E} \end{cases}$
3. $F_\rho(x) = \begin{cases} y & \text{if } x \neq \mathtt{F},\ x \neq \mathtt{E},\ \rho \text{ is applicable to } x \text{ and} \\ & y \text{ is the conclusion of the application of } \rho \text{ to } x \\ \mathtt{F} & \text{if } x \neq \mathtt{E} \text{ and } \rho \text{ is not applicable to } x \\ \mathtt{E} & \text{if } x = \mathtt{E} \end{cases}$
4. $F_{then}[d_1, d_2](x) = \begin{cases} d_2(d_1(x)) & \text{if } d_1(x) \neq \mathtt{F} \text{ and } x \neq \mathtt{E} \\ \mathtt{F} & \text{if } d_1(x) = \mathtt{F} \\ \mathtt{E} & \text{if } x = \mathtt{E} \end{cases}$
5. $F_{orelse}[d_1, d_2](x) = \begin{cases} d_1(x) & \text{if } d_1(x) \neq \mathtt{F} \text{ and } x \neq \mathtt{E} \\ d_2(x) & \text{if } d_1(x) = \mathtt{F} \\ \mathtt{E} & \text{if } x = \mathtt{E} \end{cases}$
6. $F_{try}[d](x) = \begin{cases} d(x) & \text{if } d(x) \neq \mathtt{F} \text{ and } x \neq \mathtt{E} \\ x & \text{if } d(x) = \mathtt{F} \\ \mathtt{E} & \text{if } x = \mathtt{E} \end{cases}$

$then(repeat(\text{“}falletac\text{”}), repeat(\text{“}fandeltac\text{”}))$ which eliminates all the outermost universal quantifiers ($repeat(\text{“}falletac\text{”})$) and then returns recursively the left conjunct ($repeat(\text{“}fandeltac\text{”})$). As a simple example, the Program Tactic corresponding to (2) can be used to perform the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\rightarrow (...(A_1(x_1) \wedge ...) \wedge A_{n-2}(x_{n-2})) \wedge A_{n-1}(x_{n-1})) \wedge A_n(x_n)}{\rightarrow (...(A_1(x_1) \wedge ...) \wedge A_{n-2}(x_{n-2})) \wedge A_{n-1}(x_{n-1})} \wedge E}{\vdots} \wedge E}{\rightarrow A_1(x_1) \wedge A_2(x_2)} \wedge E}{\rightarrow A_1(x_1)} \wedge E \tag{3}$$

In this case only $\wedge E$ is recursively executed. The same Program Tactic can also be used to perform the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\rightarrow \forall x_1 \forall x_2 \ldots \forall x_n((...(A_1(x_1) \wedge ...) \wedge A_{n-2}(x_{n-2})) \wedge A_{n-1}(x_{n-1})) \wedge A_n(x_n))}{\rightarrow \forall x_2 \ldots \forall x_n((...(A_1(x_1) \wedge ...) \wedge A_{n-2}(x_{n-2})) \wedge A_{n-1}(x_{n-1})) \wedge A_n(x_n))} \forall E}{\vdots} \forall E}{\rightarrow \forall x_n((...(A_1(x_1) \wedge ...) \wedge A_{n-2}(x_{n-2})) \wedge A_{n-1}(x_{n-1})) \wedge A_n(x_n))} \forall E}{\rightarrow (...(A_1(x_1) \wedge ...) \wedge A_{n-2}(x_{n-2})) \wedge A_{n-1}(x_{n-1})) \wedge A_n(x_n)} \wedge E}{\rightarrow (...(A_1(x_1) \wedge ...) \wedge A_{n-2}(x_{n-2})) \wedge A_{n-1}(x_{n-1})} \wedge E}{\vdots} \wedge E}{\rightarrow A_1(x_1) \wedge A_2(x_2)} \wedge E}{\rightarrow A_1(x_1)} \wedge E \tag{4}$$

In this case $\forall E$ first and $\wedge E$ afterwards are recursively executed. We can prove in MT the following theorem:

$$\begin{array}{l} apply(\\ \quad orelse(progress(repeat(\text{“}fandeltac\text{”})), then(repeat(\text{“}falletac\text{”}), repeat(\text{“}fandeltac\text{”}))), \\ \quad \text{“}\forall x_1 \forall x_2 \ldots \forall x_n((...(A_1(x_1) \wedge ... \wedge A_{n-2}(x_{n-2})) \wedge A_{n-1}(x_{n-1})) \wedge A_n(x_n))\text{”}) = \text{“}A_1(x_1)\text{”} \end{array} \tag{5}$$

3 A model of MT

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.

tacticals implemented in `LCF` and NuPRL [8]. Their axiomatization is given in figure 2. Intuitively, $LTac$ holds over terms that are Logic Tactics (see axioms (A14)–(A21)). *then* is used to apply tactics sequentially. It applies its first argument; if it succeeds, it applies the second, it fails otherwise. *orelse* captures failure. It applies the first tactic; if it fails, it applies the second. An alternative proof strategy can thus be applied when the first strategy fails. Notice that, in `LCF`, `ORELSE` is defined by means of `?`, i.e. `let (T1 ORELSE T2)g = T1(g) ? T2(g)` [22]. Analogously, axiom (A10) states the relation between *orelse* and Γ. *try* applies the tactic to the argument. If it fails, it returns the argument. *try* is used to apply proof strategies without having failure. *progress* applies the tactic to the argument. If the result is equal to the argument (no progress in the proof has been obtained) then it fails. *repeat* applies the tactic until the tactic fails. *repeat* can be used to express repeated applications of tactics.

Logic Tactics can be formally defined as follows. They are a subset of the terms of MT. This set is based upon the set of constants "t_ρ": $\mathcal{T}_0 = \{\text{“}t_\rho\text{”} : \rho \in \mathcal{R}\}$. From $\mathcal{T}_0$, we inductively construct the set of Logic Tactics $\mathcal{T}$.

1. “$idtac$” $\in \mathcal{T}$, “$failtac$” $\in \mathcal{T}$, $\mathcal{T}_0 \subseteq \mathcal{T}$.
2. If $t_1, t_2 \in \mathcal{T}$, then $then(t_1, t_2) \in \mathcal{T}$, $orelse(t_1, t_2) \in \mathcal{T}$, $repeat(t_1) \in \mathcal{T}$, $try(t_1) \in \mathcal{T}$, and $progress(t_1) \in \mathcal{T}$.

The following set of axioms circumscribes the set of terms that are Logic tactics.

$$
\begin{array}{ll}
(A14) & LTac(\text{“}idtac\text{”}) \\
(A15) & LTac(\text{“}failtac\text{”}) \\
(A16) & LTac(\text{“}t_\rho\text{”}), \text{ for any “}t_\rho\text{”} \in \mathcal{T}_0 \\
(A17) & \forall t_1 \forall t_2 \;\; LTac(t_1) \wedge LTac(t_2) \supset LTac(then(t_1, t_2)) \\
(A18) & \forall t_1 \forall t_2 \;\; LTac(t_1) \wedge LTac(t_2) \supset LTac(orelse(t_1, t_2)) \\
(A19) & \forall t \;\; LTac(t) \supset LTac(try(t)) \\
(A20) & \forall t \;\; LTac(t) \supset LTac(progress(t)) \\
(A21) & \forall t \;\; LTac(t) \supset LTac(repeat(t))
\end{array}
$$

As an example of Logic Tactic, consider the following:

$$
\begin{array}{l}
orelse(progress(repeat(\text{“}fandeltac\text{”})), \\
\qquad then(repeat(\text{“}falletac\text{”}), repeat(\text{“}fandeltac\text{”})))
\end{array}
\tag{2}
$$

$repeat(\text{“}fandetac\text{”})$ applies $\wedge E$ till it fails. If the result is different from the argument it has been applied to, i.e. if $progress(repeat(\text{“}fandeltac\text{”}))$ succeeds, then *orelse* terminates execution. If no progress in the proof has been made, i.e. the argument which is applied to is not a conjunction, then $progress(repeat(\text{“}fandeltac\text{”}))$ fails and *orelse* recovers from failure by executing

$(A7)$ **idtac** : $\forall x\ (Tac(x) \supset apply(\text{“}idtac\text{”}, x) = idtac(x))$

$(A8)$ **failtac** : $\forall x\ (Tac(x) \supset apply(\text{“}failtac\text{”}, x) = failtac(x))$

$(A9)$ **then** : $\forall x \forall t_i \forall t_j\ (Tac(x) \wedge LTac(t_i) \wedge LTac(t_j) \supset$
$apply(then(t_i, t_j), x) = if\ (apply(t_i, x) = fail)$
$then\ fail$
$else\ apply(t_j, apply(t_i, x)))$

$(A10)$ **orelse** : $\forall x \forall t_i \forall t_j\ (Tac(x) \wedge LTac(t_i) \wedge LTac(t_j) \supset$
$apply(orelse(t_i, t_j), x) = \Gamma(apply(t_i, x), apply(t_j, x)))$

$(A11)$ **try** : $\forall x \forall t_i\ (Tac(x) \wedge LTac(t_i) \supset$
$apply(try(t_i), x) = apply(orelse(t_i, \text{“}idtac\text{”}), x))$

$(A12)$ **progress** : $\forall x \forall t_i\ (Tac(x) \wedge LTac(t_i) \supset$
$apply(progress(t_i), x) = if\ (apply(t_i, x) = x)$
$then\ fail$
$else\ apply(t_i, x))$

$(A13)$ **repeat** : $\forall x \forall t_i\ (Tac(x) \wedge LTac(t_i) \supset$
$apply(repeat(t_i), x) = if\ (apply(t_i, x) = fail)$
$then\ x$
$else\ apply(repeat(t_i), apply(t_i, x)))$

where x, t_i and t_j are variables of MT.

Figure 2: Axiomatization of tacticals

where $f_\rho(\text{“}s\text{”})$ denotes the conclusion of the application of ρ. If ρ is not applicable to s, then we have $\vdash_{\mathrm{MT}} \neg P_\rho(\text{“}s\text{”})$, and therefore $\vdash_{\mathrm{MT}} t_\rho(\text{“}s\text{”}) = fail$ (rule $if\ E_\neg$). Finally, we have failure propagation, i.e. $\vdash_{\mathrm{MT}} t_\rho(fail) = fail$.

As examples, consider the function symbols $fandeltac$, $falletac$ and $fallitac$ which correspond to t_ρ for $\wedge E$, $\forall E$ and $\forall I$, respectively:

$$
\begin{array}{lll}
(A3_{\wedge E}) & \forall x\ (Tac(x) \supset fandeltac(x) = & if\ (\neg Fail(x) \wedge Conj(x)) \\
 & & then\ fandel(x) \\
 & & else\ fail\ ) \\
(A3_{\forall E}) & \forall x\ (Tac(x) \supset falletac(x) = & if\ (\neg Fail(x) \wedge Forall(x)) \\
 & & then\ falle(x) \\
 & & else\ fail\ ) \\
(A3_{\forall I}) & \forall x \forall y\ (Tac(x) \supset fallitac(x,y) = & if\ (\neg Fail(x) \wedge Var(y) \wedge NoFree(y,x)) \\
 & & then\ falli(x,y) \\
 & & else\ fail\ )
\end{array}
$$

Let us now consider Program Tactics that are built as compositions of simpler Program Tactics. `GETFOL`, like most tactic-based theorem provers [21, 27, 28, 8], provides the ability to combine Program Tactics by means of *tacticals*. Tacticals are implemented by control constructs that compose tactics in a principled manner. For instance, tacticals are used to control sequential and repeated applications of tactics and to handle failure in the application of tactics. Tacticals compose tactics by taking tactics as arguments and returning tactics as results. In ML, this is achieved by using a higher order syntax where tactics are passed as arguments to programs. In MT (which is first order) we obtain a similar result by adding to $\mathcal{ML}$ the constants “t_ρ”, i.e. for each function symbol $t_\rho \in \mathcal{ML}$ we have a constant “t_ρ” $\in \mathcal{ML}$. We say that “t_ρ” is the name of t_ρ. For each “t_ρ”, we have in MT an axiom stating the relation between the function symbol and its name.

$$(A4) \quad \forall x\ (Tac(x) \supset apply(\text{“}t_\rho\text{”}, x) = t_\rho(x))$$

where, $apply$ is a function symbol. Notice that “t_ρ” is a name of an object of MT rather than of OT. We have constants (“$idtac$” and “$failtac$”) and function symbols ($idtac$ and $failtac$) for trivial tacticals that return the argument unchanged and that generate failure, respectively. We also have the following axioms:

$$
\begin{array}{ll}
(A5) & \forall x\ (Tac(x) \supset idtac(x) = x) \\
(A6) & \forall x\ (Tac(x) \supset failtac(x) = fail)
\end{array}
$$

In MT, tacticals are function symbols. We call these function symbols, *Logic Tacticals*. The Logic Tacticals considered in this paper are $then$, $orelse$, try, $progress$ and $repeat$. They correspond to

GETFOL Program Tactics either construct a theorem or fail[3]. In the former case, they construct a data structure that memorizes a sequent. In the latter, they construct a data structure for failure. The basic assumption underlying computations of Program Tactics is that these data structures cannot be confused, i.e. the interpreter of Program Tactics is always able to distinguish between (a data structure memorizing) failure and (a data structure memorizing) a sequent. This fact is asserted in MT by the following axiom:

$$(A2) \quad \forall x \ \neg(Seq(x) \wedge Fail(x))$$

Moreover, a Program Tactic never asserts failure as a theorem. Analogously, in MT, from axiom (A2) and the axiom stating $\forall x(T(x) \supset Seq(x))$ we can prove

$$\neg \ T(fail) \tag{1}$$

Since Program Tactics construct either theorems or failures, they must accept as arguments either theorems or failures, and nothing more. This allows us to compose them. Being a well sorted argument for a tactic is expressed by the predicate Tac.

$$(D3) \quad \forall x \ (Tac(x) \leftrightarrow T(x) \vee Fail(x))$$

Let us consider Program Tactics that apply a single inference rule ρ. If ρ is applicable, the Program Tactic succeeds and constructs a theorem, i.e. the conclusion of the application of ρ. If ρ is not applicable, then the Program Tactic fails. If a failure is given in input, the Program Tactic that applies ρ fails, i.e. it "propagates" failure. For each $\rho \in \mathcal{R}$, we extend MT with the function symbol t_ρ and the following axiom

$$\begin{array}{ll} (A3) \quad \forall x \ (Tac(x) \supset t_\rho(x) = & if \ (\neg Fail(x) \wedge P_\rho(x)) \\ & then \ f_\rho(x) \\ & else \ fail \ ) \end{array}$$

Notice that the function symbol t_ρ corresponds intuitively to a Program Tactic that applies a single inference rule ρ. If ρ is applicable to a given premise s which is a theorem of OT, then $\vdash_{\mathrm{MT}} Tac(\text{“}s\text{”})$, $\vdash_{\mathrm{MT}} \neg Fail(\text{“}s\text{”})$ and $\vdash_{\mathrm{MT}} P_\rho(\text{“}s\text{”})$. Hence, we have $\vdash_{\mathrm{MT}} t_\rho(\text{“}s\text{”}) = f_\rho(\text{“}s\text{”})$ (rule $if \ E$),

[3] Actually GETFOL allows for both "backward tactics" (which compose backward rules, e.g. backward "conjunction left elimination") and "forward tactics" (which compose forward rules, e.g. forward "conjunction left elimination"). Roughly speaking, backward tactics are functions from goals to lists of subgoals, while forward tactics are functions from lists of theorems to theorems. In this paper, we consider only forward rules and tactics. The axiomatization of backward rules and tactics is conceptually similar, even if technically different from that of forward rules and tactics.

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

Figure 1: Conditional inference rules

Program Tactics may encode proof strategies that are not guaranteed to succeed, i.e. they may attempt to apply a rule that is not applicable. If this is the case, the Program Tactic fails. The ability of handling failure is an essential feature of programming languages (like ML) that are used to encode Program Tactics. In ML, for instance, the user can specify that, under certain conditions, a program fails and that, when a program fails, an alternative program can be tried. This is achieved in ML with the construct `?`. The value of the expression `e1 ? e2` is the value of `e1` if `e1` does not fail, otherwise it is the value of `e2`. MT must have the same capability. To obtain this, we add to the language a predicate $Fail$ which holds of failures in the application of object level inference rules. This predicate symbol corresponds to a program in `GETFOL` which implements a boolean function which returns `TRUE` when its argument computes a data structure that encodes failure. We therefore extend the language of MT with a constant, $fail$ that corresponds to the data structure for failure, and define the predicate $Fail$ in terms of $fail$ as follows:

$$(D1) \quad \forall x\ (Fail(x) \leftrightarrow x = fail)$$

For simplicity, we suppose that we have only one constant for failure[2]. We can now define a function symbol Γ

$$(D2) \quad \forall x_1 x_2 \quad \Gamma(x_1, x_2) = \begin{array}{l}\mathit{if}\ (x_1 = fail)\\ \mathit{then}\ x_2\\ \mathit{else}\ x_1\end{array}$$

which has a behaviour analogous to the ML construct `?`. Indeed, if $\vdash_{\mathrm{MT}} x_1 \neq fail$, then $\vdash_{\mathrm{MT}} \Gamma(x_1, x_2) = x_1$ (where $\vdash_{\mathrm{MT}}$ stands for provability in MT). If $\vdash_{\mathrm{MT}} x_1 = fail$, then $\vdash_{\mathrm{MT}} \Gamma(x_1, x_2) = x_2$.

[2]Program Tactics distinguish different failures depending on the tactic which fails. In ML this is achieved by means of a "failure token" passed as argument to the expression that generates failure, i.e. `failwith`. In the actual `GETFOL` code different data structures are generated depending on the tactic that fails. MT can be easily extended with a set of constants each denoting a different failure and with constructs similar to the ML constructs for "selective failure trapping", e.g. the ML construct `??` [22].

introduction rule, then we have that $\neg P_\rho(\text{“}A(x) \longrightarrow A(x)\text{”})$ is provable in MT. T is the provability predicate. We have an axiom $T(\text{“}s\text{”})$, for any sequent s which is an object level axiom ($s \in \mathcal{A}$) or assumption (of the form $A \longrightarrow A$). These axioms state in MT the provability in OT of axioms and assumptions. Therefore, $(A1)$ states that, if the rule ρ is applicable $[P_\rho(x)]$ to a provable sequent $[T(x)]$, then the conclusion $[f_\rho(x)]$ is a provable sequent $[T(f_\rho(x))]$. As an example, consider the following ND rules for left conjunction elimination ($\wedge E$), universal specialization ($\forall E$) and introduction ($\forall I$):

$$\frac{\Gamma \longrightarrow A \wedge B}{\Gamma \longrightarrow A}\ \wedge E \qquad \frac{\Gamma \longrightarrow \forall x A(x)}{\Gamma \longrightarrow A(x)}\ \forall E \qquad \frac{\Gamma \longrightarrow A(x)}{\Gamma \longrightarrow \forall x A(x)}\ \forall I$$

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

The corresponding axioms are the following:

$$\begin{array}{ll} (A1_{\wedge E}) & \forall x((T(x) \wedge Conj(x)) \supset T(fandel(x))) \\ (A1_{\forall E}) & \forall x((T(x) \wedge Forall(x)) \supset T(falle(x))) \\ (A1_{\forall I}) & \forall x \forall y((T(x) \wedge Var(y) \wedge NoFree(y,x)) \supset T(falli(x,y))) \end{array}$$

where, we write P_ρ and f_ρ as $Conj$ and $fandel$ when ρ is $\wedge E$, $Forall$ and $falle$ when ρ is $\forall E$, $Var(y) \wedge Nofree(y,x)$ and $falli$ when ρ is $\forall I$. Intuitively, $Conj$ holds over sequents whose formula is a conjunction, $Forall$ holds over sequents whose formula is universally quantified, Var holds over individual variables, and $Nofree(y,x)$ holds if y denotes a variable which does not appear free in the dependencies of the sequent denoted by x. $fandel$, $falle$ and $falli$ stand for “*f*orward *and* *l*eft *e*limination”, “*f*orward for*all* *e*limination” and “*f*orward for*all* *i*ntroduction”, respectively. Intuitively, their intended meaning is the function that, given the premises, returns the conclusion of the corresponding rule.

Finally, MT has axioms about the syntactic categories of OT, e.g. $Seq(\text{“}s\text{”})$ and $Var(\text{“}x\text{”})$ for any sequent s and individual variable x of OT, respectively, and about the relations between sequents and theorems, i.e. $\forall x\ (T(x) \supset Seq(x))$. A detailed description of the metatheory described so far and of its relation with the `GETFOL` code can be found in [17] (see also [16]).

In the following, we extend MT to be expressive enough to represent Logic Tactics which can be put in correspondence with Program Tactics. Program Tactics make extensive use of conditional constructs. We therefore extend MT's language with conditional term constructors *if* A *then* t_1 *else* t_2, where A is a wff and t_1, t_2 are terms, and its deductive machinery with the relevant elimination and introduction rules (reported in figure 1). Rule $if\ E$ $[if\ E_\neg]$ states that from $P(\textit{if}\ A\ \textit{then}\ t_1\ \textit{else}\ t_2)$ 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 $P(\textit{if}\ A\ \textit{then}\ t_1\ \textit{else}\ t_2)$ ($[A]$ denotes the fact that A is discharged). The resulting theory is a conservative extension of MT.

fails. In Section 5 we extend MT in order to express the logical analogous of programming constructs (like conditionals and environment constructors) which may be used in the definition of Program Tactics. In Section 6 we discuss the related work. Some conclusions and future developments are in Section 7. In this paper, we do not formally describe `GETFOL` Program Tactics. They are similar to the ML Program Tactics developed in other theorem provers [21, 27, 28, 8]. Moreover, we do not discuss how expressions in MT can be translated into `GETFOL` Program Tactics and vice versa. This non-trivial issue is addressed in other papers, in particular in [15, 13] and partly in [17] (for the version of MT which does not express tacticals). The proofs of the theorems are in the appendix.

2 Logic Tactics

Let OT be a first order object theory, defined as a triple $< \mathcal{L}, \mathcal{A}, \mathcal{R} >$, $\mathcal{L}$ being the language, $\mathcal{A}$ the set of axioms and $\mathcal{R}$ the set of inference rules of OT. We assume that inference rules $\rho \in \mathcal{R}$ apply to pairs $< \Gamma, A >$, written $\Gamma \rightarrow A$, where A is a formula and Γ a finite set of formulas. We call $\Gamma \rightarrow A$, a sequent, A the formula of $\Gamma \rightarrow A$ and Γ the dependencies of $\Gamma \rightarrow A$. We also call sequents of the form $A \rightarrow A$, assumptions. We assume that $\mathcal{R}$ contains rules which are a sequent version of the Natural Deduction (ND) rules [30, 16, 17]. (ND is the logic of `GETFOL`.) Nevertheless, the work described in this paper is largely independent of the specific inference rules considered. We take the notion of deduction defined in [30]. We call a deduction of a formula A depending on the possibly empty set Γ of formulas, a proof of the sequent $\Gamma \rightarrow A$. We say that $\Gamma \rightarrow A$ is a theorem of OT, or that $\Gamma \rightarrow A$ is provable in OT, iff there exists a proof in OT of $\Gamma \rightarrow A$.

We define a distinct first order logical theory MT $= < \mathcal{ML}, \mathcal{MA}, \mathcal{MR} >$ to be the metatheory of OT. The set of rules of MT, $\mathcal{MR}$ includes ND rules and rules for equality. We fix a naming relation between MT and the objects of OT, i.e. $\mathcal{ML}$ contains names of sequents, wffs and terms of OT. For instance, the constants "$\Gamma \rightarrow \forall x A(x)$", "$\forall x A(x)$" and "$x$" are the names of the sequent $\Gamma \rightarrow \forall x A(x)$, of the wff $\forall x A(x)$ and of the variable x, respectively. For each object level n-ary inference rule ρ, $\mathcal{ML}$ has a n-ary function symbol (that we write as f_ρ), and a n-ary predicate symbol (that we write as P_ρ). (For simplicity, in the general description of MT, we consider unary inference rules only. As shown by the examples, the extension for n-ary rules is trivial.) For each $\rho \in \mathcal{R}$, $\mathcal{MA}$ has the following axiom:

$$(A1) \quad \forall x((T(x) \wedge P_\rho(x)) \supset T(f_\rho(x)))$$

MT has axioms about f_ρ and P_ρ. For instance, if ρ is (a form of) universal specialization that replaces the outermost universally quantified variable with a free variable, we have that $f_\rho($"$\Gamma \rightarrow \forall x A(x)$"$) =$ "$\Gamma \rightarrow A(x)$", $P_\rho($"$\Gamma \rightarrow \forall x A(x)$"$)$ and $\neg P_\rho($"$\Gamma \rightarrow A(x)$"$)$ are theorems of MT. This intuitively means that the application of ρ to $\Gamma \rightarrow \forall x A(x)$ gives $\Gamma \rightarrow A(x)$, and that ρ is applicable to $\Gamma \rightarrow \forall x A(x)$, and not to $\Gamma \rightarrow A(x)$. P_ρ may express inference rule restrictions. For instance, if ρ is the universal

The importance of having tactics as expressions of a logical language (property 1.) has been largely recognized in the literature (see for instance [2, 6, 9, 11, 19, 23, 26, 31]). First, the logic can be used to reason about proof strategies [6, 19, 31] and to reduce the burden of establishing their correctness [23]. Second, a declarative account of proof strategies can be used to build new strategies, e.g by proof planning [6], by analogy [26] and by proof reuse [10].

The link between Logic and Program Tactics (property 2.) is motivated by the desire of using logical deduction to automatically and safely synthesize/optimize Program Tactics, and to reason about and extend/modify the `GETFOL` system code. The work presented in this paper is part of a long term project whose ultimate goal is to implement a provably correct theorem prover (the idea is to use `GETFOL` to proof check correctness statements about its own code). This work builds on some first significative results presented in [17]. In [17] we have proposed a metatheory (also called MT) where:

(a) There is a precise correspondence between certain wffs (called in [17], "primitive tactics") of the metatheory and the code implementing primitive Program Tactics in `GETFOL`, i.e. possibly failing applications of basic inference rules.

(b) It is possible to prove wffs (called in [17], "tactics") which specify how to build *finite sequential compositions* of primitive tactics.

(c) It is possible to give tactics a procedural content, i.e. to use them to assert object level theorems, either by *interpreting* or *compiling* them into `GETFOL` code.

In this paper, we extend MT to be expressive enough to represent the kind of Program Tactics used in most tactic-based interactive theorem provers (*e.g.* [21, 27, 28, 8]). We axiomatize the most interesting tacticals, *i.e. then*, *orelse*, *try*, *progress* and *repeat*. Tacticals provide a powerful and well tested mechanism for controlling proof search. As a consequence, MT can be used to express useful and complex tactics. In particular, the axiomatization of the tactical *repeat*, the standard tactical used to write strategies based on recursive applications of tactics, is very important. A single Logic Tactic constructed using *repeat*, in [17] may have to be expressed with an infinite number of tactics. Moreover, Logic Tactics constructed through *repeat* may correspond to Program Tactics which are not guaranteed to terminate, e.g. Program Tactics which do not terminate when applied to certain arguments. This kind of Program Tactics may still be very useful, i.e. they can be actually used to prove theorems in the cases when they terminate. In this paper, we show that this kind of Logic Tactics is safely represented, i.e. MT is consistent. Notice that, in most interactive theorem provers (see for instance [3]) the user is required instead to prove the termination of recursive definitions.

The paper is structured as follows. In Section 2 we define Logic Tactics. In Section 3 we construct a model of MT, thus proving its consistency. We also discuss some of the requirements that this model imposes on Program Tactics. In Section 4 we prove some interesting properties of Logic Tactics, namely that deduction in MT can be used to prove when a tactic succeeds and when it

Program Tactics and Logic Tactics*

Fausto Giunchiglia[1,2] and Paolo Traverso[1]
[1]IRST - Istituto per la Ricerca Scientifica e Tecnologica
38050 Povo, Trento, Italy
[2]University of Trento, Via Inama 5, 38100 Trento, Italy
fausto@irst.it leaf@irst.it

Abstract

In this paper we present a first order classical metatheory, called MT, with the following properties: (1) tactics are terms of the language of MT (we call these tactics, *Logic Tactics*); (2) there exists a mapping between Logic Tactics and the tactics developed as programs within the GETFOL theorem prover (we call these tactics, *Program Tactics*). MT is expressive enough to represent the most interesting tacticals, *i.e. then*, *orelse*, *try*, *progress* and *repeat*. *repeat* allows us to express Logic Tactics which correspond to Program Tactics which may not terminate. This work is part of a larger project which aims at the development and mechanization of a metatheory which can be used to reason about, extend and, possibly, modify the code implementing Program Tactics and the GETFOL basic inference rules.

1 Introduction

GETFOL [12] [1] is a tactic-based interactive theorem prover. In GETFOL, tactics can be developed as programs of the GETFOL programming language [14, 12, 16]. These kinds of tactics are conceptually similar to the tactics developed in ML [22] and used in LCF and its descendants [21, 27, 28, 8]. We call these tactics, *Program Tactics*.

This paper describes a first order classical metatheory, called MT, with the following properties:

1. Tactics are terms of the language of MT. We call these tactics, *Logic Tactics*.
2. There exists a precise correspondence between Logic Tactics and Program Tactics.

*Some parts of this paper are preliminarly discussed in a paper appeared in the proceedings of the Fifth International Conference on Logic Programming and Automated Reasoning (LPAR'94).

[1] GETFOL has been developed on top of a reimplementation of the FOL system [32].

ISTITUTO PER LA RICERCA SCIENTIFICA E TECNOLOGICA
I 38100 TRENTO – LOC. PANTÉ DI POVO – TEL. 0461–814444
TELEX 400874 ITCRST – TELEFAX 0461–810851

PROGRAM TACTICS AND LOGIC TACTICS

Fausto Giunchiglia
Paolo Traverso

January 1993
Technical Report # 9301-01

Publication Notes: In Proceedings "LPAR'94, 5th International Conference on Logic Programming and Automated Reasoning", Kiev, Ukraine, July 16-21, 1994. Also presented at the "Third International Symposium on Artificial Intelligence and Mathematics", Fort Lauderdale, Florida, January 1994.

ISTITUTO TRENTINO DI CULTURA