Reflective reasoning with and between a declarative metatheory and the implementation code

Fausto Giunchiglia[1,2] and Paolo Traverso[1]
1IRST - Istituto per la Ricerca Scientifica e Tecnologica
38050 Povo, Trento, Italy
[2]DIST, University of Genoa, Via Opera Pia 11A, Genova, Italy
fausto@irst.it leaf@irst.it

Abstract

The goal of this paper is to present a theorem prover where the underlying code has been written to behave as the procedural metalevel of the object logic. We have then defined a logical declarative metatheory MT which can be put in a one-to-one relation with the code and automatically generated from it. MT is proved correct and complete in the sense that, for any object level deduction, the wff representing it is a theorem of MT, and viceversa. Such theorems can be translated back in the underlying code. This opens up the possibility of deriving control strategies automatically by metatheoretic theorem proving, of mapping them into the code and thus of extending and modifying the system itself. This seems a first step towards "really" self-reflective systems, *it.* systems able to reason deductively about and modify their underlying computation mechanisms. We show that the usual logical reflection rules (so called reflection up and down) are derived inference rules of the system.

1 Introduction

Reflective and metatheoretic reasoning are well known techniques applied in knowledge representation and automated deduction (see for instance [Bundy, 1988 , [Constable *et* al., 1986], [Bowen and Kowalski, 1982], [Smith, 1983], [Gordon *et* al., 1979]). Roughly speaking, in the past, metareasoning has been performed according to two different paradigms. In the first, from now on called procedural, the metalevel consists of a programming language and metareasoning is performed by *computation* in it. One example in AI is [Smith, 1983], another in theorem proving is LCF and its metalanguage ML [Gordon *et* al., 1979]. In LCF the user can write control strategies as programs (usually called *tactics)* in ML to guide the search for a proof of a theorem. In the second paradigm, from now on called declarative, the metalevel is a logical metatheory and metareasoning is performed by *deduction* on metalevel statements. One example in AI is [Weyhrauch, 1980], one in theorem proving is [Howe, 1988]. Both approaches are sometimes incorporated and alternatively used; thus, for instance, in NuPrl [Constable *et* al., 1986] and Isabelle [Paulson, 1989] both ML and a declarative logical metatheory can be used to build derived inference rules. In logic programming, metainterpreters [Bowen and Kowalski, 1982] can be seen both procedurally and declaratively.

In this paper, we present a system (called GETFOL [1]*)* with both a procedural and a declarative metalevel. In this respect GETFOL is similar to NuPrl and Isabelle; on the other hand GETFOL has features which make it very different from any other system proposed so far:

(1) the metalevel programming language is the same as the underlying implementation language and the code implementing the object logic has been written to be its procedural metalevel.

(2) the logical declarative metatheory MT can be put in a one-to-one relation with the code and automatically generated from it (and viceversa).

(3) MT is correct and complete in the sense that, for any object level deduction (performed by running the code implementing the object level logic), the wff representing it is a theorem of MT[2]. Such theorems, possibly proved automatically by metatheoretic theorem proving, can be "mapped back" into the underlying code as new reasoning modules. These modules, if executed, will produce the proof represented by the theorem they have been translated from.

As a consequence of these three facts, it is possible to generate (parts of) MT automatically from the implementation language, to prove in MT "certain" theorems and then to "transform" them into new code. The result is an extension or, possibly, a modification of the system itself. The GETFOL underlying code is not a "black box", fixed once and for all at the time of the development, but can change over time. This seems a first step towards "really" self-reflective systems, *it.* systems able to reason deductively about and thus, possibly, modify, their

[1] GETFOL is a reimplementation/ extension of the FOL system [Weyhrauch, 1980]. GETFOL has, with minor variations, all the functionalities of FOL plus extensions, some of which described here, to allow metatheoretic theorem proving.

[2]The notions of correctness and completeness here involved are sometimes called adequacy and faithfulness, respectively.

underlying reasoning strategies. As a side effect of this "reflective" relation existing between computation and deduction, the usual logical reflection rules (reflection up and reflection down) [Giunchiglia and Smaill, 1989] can be proved to be (a form of) derived inference rules.

The paper is structured as follows. Section 2 describes how the implementation has been constructed to behave as the procedural metalevel of the system. Section 3 describes MT and how it can be automatically generated from the implementation code. In section 4, it is proved that MT represents all the object level deductions and that deduction in MT is the analogous operation of writing tactics in the implementation language. This is the fundamental property that allows the interpretation of theorems of MT in terms of the underlying code (section 5) via the use of reflection up and down. Finally, section 6 gives some conclusions and a short discussion of the related work.

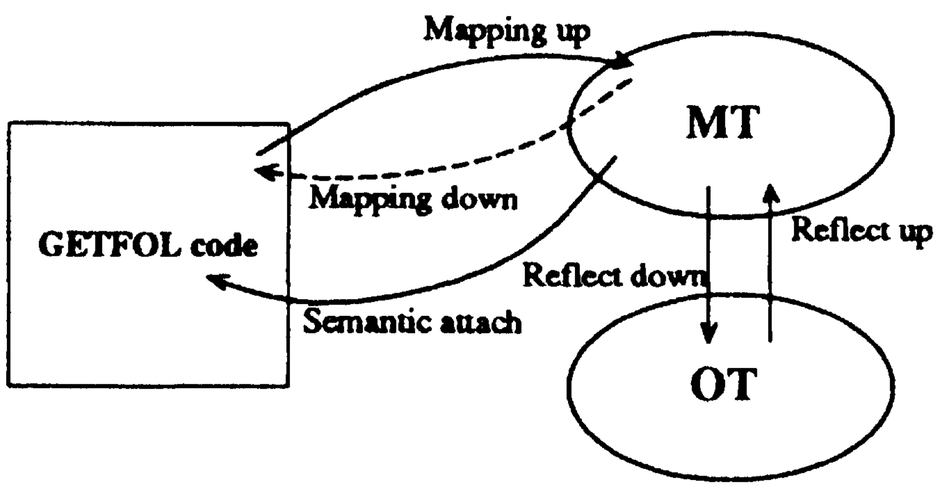

Figure 1: The GETFOL system.

2 The system code as the procedural metalevel

GETFOL allows the definition of multiple distinct theories. Each theory is formally defined as a triple ⟨Language, Axioms, Set of inference rules⟩. To simplify things we consider here the case where we have only one object theory $\mathbf{OT}=\langle \mathcal{L}, \mathcal{A}x, \mathcal{R}\rangle$ and one metatheory $\mathbf{MT}=\langle \mathcal{ML}, \mathcal{MA}x, \mathcal{R}\rangle$ (see figure 1). MT and OT use a first order classical sequent logic. By *sequent* we mean here a pair (Γ, A), written also $\Gamma \vdash A$, where A is a formula and Γ a set of formulas. For simplicity, in this paper we suppose MT and OT use the same set of inference rules $\mathcal{R}$. The inference rules, which are a sequent version of Prawitz' natural deduction (ND) calculus [Weyhrauch, 1980], allow introduction and elimination only in the post-sequent A. In the following, to simplify notation, when not relevant, we write A for $\Gamma \vdash A$. Thus, for instance

$\wedge$E $\dfrac{A \wedge B}{A}$ $\forall$I $\mathbf{x_1}$ $\mathbf{x_2}$ $\dfrac{A(x_2)}{\forall x_1 A(x_1)}$ $\forall$E **x t** $\dfrac{\forall x A(x)}{A(t)}$

are respectively one of the two conjunction elimination rules ($\wedge E$) and the universal quantifier introduction and elimination rules ($\forall I$,$\forall E$). The functions in the underlying **GETFOL** code implementing these inference rules are given in figure 2. Thus, for instance, when applicable, (**alli-fun** $A(x_2)$ x_1 x_2) returns $\forall x_1 A(x_1)$ which is then added (by proof-add-theorem) to the current proof. Notice that, even if the underlying code has been written to treat errors *(tg.* the application of *(AE)* to a disjunction), this issue is not dealt with in this paper.

```
(DEFLAM ande (X)
 (IF (AND (IS-A-THEOREM X) (CONJ X))
  THEN (proof-add-theorem (ande-fun X))))

(DEFLAM alli (X1 X2 X3)
 (IF (AND (AND (IS-A-THEOREM X1)
          (AND (IS-A-VAR X2) (IS-A-VAR X3)))
          (NO-FREE X3 X1))
  THEN (proof-add-theorem (alli-fun X1 X2 X3))))

(DEFLAM alle (X1 X2 X3)
 (IF (AND (AND (IS-A-THEOREM X1)
          (AND (IS-A-VAR X2) (IS-A-TERM X3)))
          (FORALL X1))
  THEN (proof-add-theorem (alle-fun X1 X2 X3))))
```

where IS-A-VAR and IS-A-TERM evaluate to TRUE if the argument is a variable or a term of the object logic, IS-A-THEOREM evaluates to TRUE if the argument is an asserted theorem in the proof and proof-add-theorem adds its argument to the proof.

Figure 2: GETFOL implementation of Ae, VJ and *VE.*

Strategies and derived inference rules can be defined in a metalevel functional language, GET, which is a subset of the implementation language of GETFOL. To preserve correctness, the development environment is such that the user can write tactics which fail but that never assert a non-theorem. For instance, considering the code in figure 2, proof-add-theorem, ande-fun, alli-fun and alle-fun are not available to the user. In this respect, GET is similar to ML as used in LCF, Nuprl or Isabelle. For instance, the GET code implementing the (simple) derived inference rule that corresponds to the deduction

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

is:

```
(DEFLAM all-distr-con (x1 x2 x3)
 (alli (ande (alle x1 x2 x3)) x2 x3))
```

Strategies can be defined by using conditionals and iterations:

```
(DEFLAM strategy1 (x1 x2 x3)
 (IF (AND (FORALL x1) (CONJ (alle x1 x2 x3)))
  THEN (all-distr-con x1 x2 x3)
  ELSE (repeat 'tactic2 x1 x2 x3)))
```

where **FORALL** and **CONJ** are built-in **GETFOL** predicates which evaluate to **TRUE** when their argument is, respectively, a universally quantified formula and a conjunction, (**repeat 'tactic args**) iterates the application of **tactic** over **args** as many times as possible. **strategy1** can be described as: "if **x1** is a universally quantified conjunction, then derive the first conjunct

of x1, otherwise exhaustively apply tactic2", where tactic2 is a previously defined tactic.

What has been described so far suggests that GET can be used as the procedural metalevel of GETFOL, analogously to what happens with ML in LCF, NuPrl or Isabelle, or even with metainterpreters in Prolog. This is, in fact, the case when writing tactics. The difference comes from the fact that (exploiting that GET is also GETFOL's implementation language), the built-in GET functions to perform logic inference are exactly those used to implement the basic inference rules, *eg.* those in figure 2. More generally, *all the* GETFOL *code has been carefully written to allow the identification of the procedural metalanguage with the implementation language.* Thus, for instance, the implementation provides GET with all the syntax manipulation routines (such as FORALL, CONJ, ande), with all the proof manipulation routines (such as IS-A-THEQREM), with all the theory manipulation routines and so on. Not only must the code produce extensionally the right behaviour (satisfying the usual correctness criteria), but it must also be written to be at the same time the *procedural metalevel of the logic it implements.* In other words, it must have the right function and predicate symbols, the internal structure of the function and predicate definitions must be such that they can be put in a one-to-one relation with the axioms describing their behavior (all of this is described in section 3), and far harder, it must be such that computation can be directly mapped into the metathereotic "representation" of the deduction it produces (described in section 4).

We call the production of code satisfying the requirements above, "the mechanization of the logic" (to distinguish it from the process of producing an implementation of the logic). Mechanizing a logic is far harder than implementing it. On the other hand, as the rest of the paper will show, the mechanization of the logic can be exploited to build really self-reflective systems, *it.* systems able to reason deductively about and modify their computation mechanisms. In fact it becomes then possible to generate automatically a logical metatheory MT from the code and, viceversa, to compile certain theorems of MT as system implementation code. Modification of the system's underlying computation mechanism is achieved by re-writing already existing (parts of) procedures. Note that this is not done in any of the existing theorem provers or metainterpreters. From this perspective, the work which most closely resembles ours is Brian Smith's [Smith, 1983]. The fundamental difference is that our metalanguage is a logical metatheory; this allows us to generate *provably correct* computation procedures *automatically* by metatheoretic theorem proving.

3 The declarative metatheory MT

MT's set of inference rules is fixed, being 7v. We need to define MT's language *MC* and axioms *MAx. MC,* has names for the elements of OT (axioms and assumptions *s,* formulas *w,* variables x and so on); this is achieved by having for any such element an individual constant ("s", "w", "*x" and so on*) as part *of MC.* These constants are the "quotation mark names" [Giunchiglia and Traverso, 1990] of the objects of OT.

[(DEFLAM <fun> (X1 ... Xn) <body>] =
∀ [X1] ... ∀ [Xn] ([<body>])
[(IF <t1> THEN <t2>)] = [<t1>] → [<t2>]
[(IF <t1> THEN <t2> ELSE <t3>)] =
***if* [<t1>] *then* [<t2>] *else* [<t3>]**
[(AND <t1> <t2>)] = [<t1>] ∧ [<t2>]
[(IS-A-THEOREM <t>)] = T([<t>])
[(proof-add-theorem <t>)] = T([<t>])
[(CONJ X1)] = $Conj$([X1])
[(FORALL X1)] = $Forall$([X1])
[(NO-FREE X1 X2)] = $NoFree$([X1],[X2])
[(IS-VAR X1)] = Var([X1])
[(IS-TERM X1)] = $Term$([X1])
[(ande-fun X1)] = $ande$([X1])
[(alli-fun X1 X2 X3)] = $alli$([X1],[X2],[X3])
[(alle-fun X1 X2 X3)] = $alle$([X1],[X2],[X3])
...
[X1] = x_1
...
[Xn] = x_n
...
where functions and predicates in $\mathcal{ML}$ (for instance *Forall*, *NoFree*, *Var*, *Term*, *ande*, *alli*, *alle* and so on) intuitively represent the computational operations performed by the code they are mapped from. The T predicate indicates "theoremhood".

Figure 3: m_{up} - mapping from the code to the metatheory.

Each inference rule of OT gets "mapped up" to a distinct axiom of *MAx.* This mapping, called *"m_{up}",* performs a one-to-one translation from the code into elements of MT (see figure 3, $rn_{up}(x)$ is written as *[x])* Thus, for instance, the axioms generated by applying m_{up} to the GET code implementing *AE,* V7 and *VE* (see figure 2) are listed in figure 4. A complete definition of rn_{up} and the code over which it can operate is outside the goals of this paper. The important point to notice is the fact that m_{up} (and its inverse) can be implemented to do the translation in either direction automatically.

$$(\mathcal{A}_{\wedge\mathcal{E}l}): \quad \forall x(T(x) \wedge Conj(x) \rightarrow T(ande(x)))$$
$$(\mathcal{A}_{\forall I}): \quad \forall x_1 \forall x_2 \forall x_3 (T(x_1) \wedge Var(x_2) \wedge Var(x_3) \wedge NoFree(x_3, x_1) \rightarrow T(alli(x_1, x_2, x_3)))$$
$$(\mathcal{A}_{\forall E}): \quad \forall x_1 \forall x_2 \forall x_3 (T(x_1) \wedge Var(x_2) \wedge Term(x_3) \wedge Forall(x_1) \rightarrow T(alle(x_1, x_2, x_3)))$$

Figure 4: Metatheoretic axioms mapped from the code.

We can observe in figure 3 that the two GET operations of testing if something is a theorem already asserted in the object theory (performed by IS-A-THEOREM) and of asserting a proved theorem (performed by proof-add-theorem) are translated into the same metatheoretic predicate T[3]. This is because OT, as

[3] Notice that this seems to contradict our previous state-

represented in MT, is the minimal set closed under the application of the inference rules to the axioms. Therefore, because MT is a metatheory of the transitive closure of OT, the procedural difference between a wff "being asserted" as a theorem and a wff "to-be asserted" as a theorem is lost.

The assertion of the axioms generated by m_{up} requires adding function and predicate symbols to $\mathcal{ML}$: for each inference rule of OT we have a function symbol in $\mathcal{ML}$ with the appropriate arity, for instance *ande*, *alli* and *alle* for $\wedge E$,$\forall I$ and $\forall E$ respectively. $\mathcal{ML}$'s predicates are $Conj$, $Forall$, $Nofree$, Var, $Term$ and so on.

MT's logic allows the use of the construct *if p then* t_1 *else* t_2, where p is a wff and t_1, t_2 are terms. The *if* construct, very important in order to make the MT's axiomatization mirror very closely the underlying code, is actually not first order. On the other hand it can be easily proved that *if* can be defined in a conservative extension of classical ND. GETFOL allows the use of the *if* construct and has an introduction and an elimination inference rule for it.

4 Deduction in MT versus program definition

The goal of this section is to prove that, given a mapping between wffs in MT and deductions in OT, *for any object level deduction (performed by running the code implementing the object level logic), the corresponding wff is a theorem of the metatheory and viceversa.* To do this we need to define how object level proofs are represented in the metatheory. This is achieved by defining a mapping **w** from sequent trees Π in OT to wffs of MT. Sequent trees are trees of sequents, each labeled by an inference rule. Sequent tree leaves are axioms or assumptions (sequents of the form $A \vdash A$). Sequent trees are not necessarily deductions as the rule labelling a sequent may not be applicable. We say that Π is a sequent tree of s if s is the endsequent of Π. The endsequent is the "root" of the sequent tree. The formulae mapped from sequent trees are called **sequent tree formulas** (in short **twffs**) and are of the form $\mathcal{P} \rightarrow T(t)$, where $\mathcal{P}$ are the **preconditions** and t the **sequent tree term** of the twff. $\mathcal{P}$ and t are inductively defined over the complexity of Π as follows:

Base: if $\Pi = s$, then $\mathbf{w}(\Pi) = T(\text{“}s\text{”})$;

Step: as examples, let us consider the cases of $\wedge E$ and $\forall I \xi \eta$. Let $\mathbf{w}(\Pi_1) = \mathcal{P}_1 \rightarrow T(t_1)$ be a twff. If Π is a sequent tree built from Π_1 by adding the rule label $\wedge E$, then $\mathbf{w}(\Pi) = \mathcal{P}_1 \wedge Conj(t_1) \rightarrow T(ande(t_1))$. If the added label is $\forall I \xi \eta$, then $\mathbf{w}(\Pi) = \mathcal{P}_1 \wedge Var(\text{“}\xi\text{”}) \wedge Var(\text{“}\eta\text{”}) \wedge NoFree(t_1, \text{“}\eta\text{”}) \rightarrow T(alli(t_1, \text{“}\xi\text{”}, \text{“}\eta\text{”}))$.

Notice that there is an isomorphism between twffs and the sequent trees they represent. In particular the sequent tree term records the tree of applications of inference rules while the preconditions record precisely the tests which allow the applications of inference rules.

Now we concentrate on showing that MT (which can be proved consistent) has the desired properties[4]:

Theorem 1 (MT correct and complete for OT) *: Let* Π *be any sequent tree of* s *in OT. Let* t *be the sequent tree term of the twff* $\mathbf{w}(\Pi)$*. Then* $\vdash_{MT} T(t) \iff \Pi$ *is a proof of* s*.*

Proof [Hinted]: ($\Leftarrow$): corollary of theorems 2 and 3 below.

($\Rightarrow$): similar to the proof of theorem 3. Q.E.D.

Theorem 1, guarantees that, for any provable $T(t)$, there is a proof of a theorem in the object theory and viceversa [5]. As a consequence, given a GET program that succeeds in building an OT proof, theoremhood applied to a term corresponding to the proof itself can be proved in MT. Notice that theorem 1 considers only derived inference rules. This result can be generalized to deal with an extension of MT, expressive enough to represent tactics, written as programs with conditionals, iterations, failure detection and so on. This non-trivial issue, which assures a complete translation of GET into MT and viceversa is not discussed here.

Theorem 1 does not tell us anything about how to prove $T(t)$, in other words, about how to construct in MT the derived inference rules. A possible technique is suggested by the following two theorems.

Theorem 2 (Twff provability) *For any sequent tree formula* $\mathcal{P} \rightarrow T(t)$, $\vdash_{MT} \mathcal{P} \rightarrow T(t)$.

Proof[hinted]: The proof is performed by induction over the complexity of Π. As an example of step case, let us consider $\wedge E$. Let us suppose that the induction hypothesis is $\mathcal{P} \rightarrow T(t)$. From it we can derive that $\mathcal{P} \vdash T(t)$. From the instantiation of axiom ($\mathcal{A}_{\wedge \mathcal{E}l}$) in figure 4 we obtain $T(t) \wedge Conj(t) \rightarrow T(ande(t))$, from which we can then derive (also considering the induction hypothesis) $\mathcal{P} \vdash Conj(t) \rightarrow T(ande(t))$. For the deduction theorem we derive thus $\mathcal{P} \rightarrow (Conj(t) \rightarrow T(ande(t)))$ which is equivalent to $\mathcal{P} \wedge Conj(t) \rightarrow T(ande(t))$. Q.E.D.

The proof suggests how twffs can be deduced in the metatheory: basic twffs can be obtained directly as instantiations of the axioms in figure 4; complex twffs can then be composed out of simpler ones following the steps hinted in the proof. This is not the only way to build twffs, [Giunchiglia and Traverso, 1990] describes some examples in detail.

The fact that all the twffs are provable in MT is exactly what we should have expected. In fact, any twff corresponds to a program that can be defined by the user in the system code. To say that any twff can be derived in the metatheory is equivalent to say that any strategy can be written in the code. *Derivation of a theorem in MT is the analogous operation of writing code in the implementation language.* Notice that this suggests a new way to develop tactics: instead of coding them in

---

ment that the m_{up} is one-to-one. In certain cases (T is one of these) m_{up} and its inverse distinguish between occurrences (in this case, corresponding either to IS-A-THEOREM or to proof-add-theorem).

[4] For lack of space, only proof outlines are given.

[5] If not desiderable, completeness can be easily dropped by generating MT only partially; for instance, we may not have the names of all the objects in OT.

the procedural metalanguage the user can theorem prove them in MT. The user can thus write the hardest steps and, interactively, generate tactics by theorem proving. This amounts to giving the user the possibility to derive not only the object level proofs but also the tactics (this idea has some resemblance with the work on proof planning [Bundy, 1988], see [Giunchiglia and Traverso, 1990] for a more in depth discussion).

A different matter is whether the strategy is successful. In the programming language, a defined strategy may generate an object level proof or fail. Similarly, the sequent tree may or may not be a proof. For instance, a twff whose sequent tree term is *ande("A AB")* corresponds to a proof, but that whose term is *ande("A VB")* does not. In metalevel programming languages (like ML [Gordon *et al.*,1979]), given two simple tactics corresponding to the ones above, they need to be executed in order to know that the former succeeds whereas the latter fails. In MT, *the derivabiliiy of preconditions determines whether twffs correspond to proofs:*

Theorem 3 (Preconditions) : *Let* Π *be any sequent tree of s in OT. Let* $\mathcal{P}$ *be the preconditions of the twff* $\mathbf{w}(\Pi)$. *Then* $\vdash_{MT} \mathcal{P} \iff \Pi$ *is a proof of s.*

Proof[hinted]. The proof is performed over the complexity of Π. As an example of the step case let us consider $\wedge E$. Let Π be:

$$\wedge E \ \frac{\begin{matrix}\Pi_1 \\ s_1\end{matrix}}{s}$$

Let $\mathbf{w}(\Pi_1)$ be $\mathcal{P}_1 \rightarrow T(t_1)$.

($\Rightarrow$) If Π is a proof then $\wedge E$ is applicable and the wff of s_1 is a conjunction. This implies $\vdash_{MT} Conj(t_1)$. By the induction hypothesis we conclude $\vdash_{MT} \mathcal{P}_1$. Then as $\mathcal{P}$ is $\mathcal{P}_1 \wedge Conj(t_1)$ we have $\vdash_{MT} \mathcal{P}$.

($\Leftarrow$) If $\vdash_{MT} \mathcal{P}_1 \wedge Conj(t_1)$, then by induction hypotheses we prove Π_1 is a proof. We prove that $\vdash_{MT} Conj(t_1)$ implies that s_1 is a conjunction. Then the $\wedge E$ rule is applicable and Π is a proof. Q.E.D.

Notice that, because of theorem 3, the success of tactics can be stated without executing them.

5 Metatheory interpretation as system code execution

Theorem 1 guarantees that, for any successful GET strategy, the corresponding twff can be deduced. The idea is to map any twff into GET code and use it to prove the goal. This idea of mapping back can be seen in two ways:

(1) The inverse of m_{up} can be defined and used to compile twffs back in the code (the "mapping down" arrow in figure 1). For any twff, the result is a strategy available to the user.

(2) Twffs can be interpreted in terms of the GET code. The result is the assertion of the sequent as a theorem in OT.

In the remainder of the paper we concentrate on the interpretation of twffs. In $\mathbf{w}(\Pi) = \mathcal{P} \rightarrow T(t)$ we can distinguish three parts: the preconditions P, the predicate *T* and the sequent tree term *t*. The assertion of the theorem in OT can be seen as the sequence of three steps: (i) prove *V* and obtain *T(t)* (subsection 5.1), (ii) from *t* generate the name of the endsequent *s* of II, "s" (not described as very similar, in principle, to step (i)), finally (iii), from T (V) assert *s* in OT (subsection 5.2).

5.1 Proving *V*

Theorem 3 tells us that the preconditions of any twff representing an object level proof can be proved by theorem proving in MT. This is the "usual" approach taken so far in theorem proving. A problem with this approach may be the size of the search space in MT which, even dropping the completeness requirement, can explode when complex metareasoning is required. The correspondence existing between MT and the GET code provides us with an alternative technique for proving in MT facts about OT. The idea is to avoid the explicit axiomatization of parts of OT and to perform computation instead of deduction. As shown above, having a mechanization of the logic gives us a one-to-one mapping between elements of the signature of MT and GET functions (section 3, figure 3) and opens up the possibility to see deductions in MT in terms of computation in GET (section 4, theorems 1,2). For how twffs are defined, their syntactic structure explicitly resembles the structure of the computation tree they represent. As a consequence we can compute in GET following the syntax of the twff. Let us consider, as an example, the case where one of the conjuncts of P is *Conj("A∧B")*. *Conj* has been mapped up from the code function CONJ and *"A∧B"* is the metatheoretic constant which denotes the theorem *A∧B*. Executing (CONJ *A∧B)* gives TRUE, this means that the metatheoretic sentence *Conj("A∧B")* is true and can be rewritten to the constant for truth, *True.*

Notice that, in order to implement the machinery described above, GETFOL must keep track of the link between the functions and predicates of the signature of MT and the GET functions they have been mapped from (by m_{up}). It must also remember which elements of OT the constants in *MC* are names of. GETFOL has in its code a data structure where it memorizes the pairs ("o",o), where "o" is an MT constant, name of o, a syntactic object in OT. The possibility to create pairs (name, object) is implemented by the *semantic attachment* functionality [Weyhrauch, 1980] shown in figure 1.

Let us give the formal definition of the interpreter implemented in GETFOL, I, which maps twffs into computation. Let us restrict ourselves to terms and atomic wffs. Let us suppose that, for any object in OT, "o" is a constant in MT. Then *I* can be defined as follows [6]:

$$\mathcal{I}(\text{“}o\text{”}) = o$$
$$\mathcal{I}(g(o_1, ..., o_p)) = \mathcal{I}(g)(\mathcal{I}(o_1), ..., \mathcal{I}(o_p))$$
$$\mathcal{I}((h_m \diamond ... \diamond h_1)(o_1 ... o_p)) = (\mathcal{I}(h_m) \diamond ... \diamond \mathcal{I}(h_1))(\mathcal{I}(o_1), ..., \mathcal{I}(o_p))$$

[6] “$\diamond$” means function composition. The notation should be made precise, by explaining how to denote function composition with functions with more than one argument. Since not relevant in this context, this issue is not faced.

Thus, for instance, instead of proving infinitely many metatheoretic theorems of the form $Conj(\text{“}A_i \wedge B_i\text{”})$, we can apply $\mathcal{I}$ to $Conj(\text{“}A_i \wedge B_i\text{”})$ to obtain $\mathcal{I}(Conj(\text{“}A_i \wedge B_i\text{”})) = \mathcal{I}(Conj)(\mathcal{I}(\text{“}A_i \wedge B_i\text{”})) = \mathcal{I}(Conj)(A_i \wedge B_i) =$ (CONJ $A_i \wedge B_i$)[7].

Notice that for any atomic wff or term *wt*, *I(wt)* is the execution of the code *c* such that $rn_{up}(c) = wt$.

Supposing that, for any n-ary function and predicate symbol /p, *fp* computes the right extension (with n-ary function symbols, the *(n* + l)-element of their set theoretic definition; with predicate symbols, either TRUE or FALSE) then J performs exactly the interpretation of terms and atomic wffs in a first order model. This result can be generalized to twffs and, more in general, to any sentence in MT. Thus the correctness and completeness of this translation of deduction into computation can be proved from the correctness and completeness results for first order logic[8].

What said above amounts to saying that OT is the standard model for MT. This is, we think, a correct way to see things and very much in agreement with Tarski's original definition of interpretation [Tarski, 1956]. More on seeing interpretation, from a computational point of view (in terms of the recursive definition of J), as the process of extracting objects from (quotation-mark and structural- descriptive) names is in section 4 of [Giunchiglia an d Traverso, 1990]).

5.2 MT-OT interaction via reflection

We can prove the following lemma:

Lemma 1 (Name equality) *Let* Π *be any sequent tree of s in OT. Let* “*s*” *be the quotation mark name of s. Let t be the sequent tree term of the twff* **w**(Π). *Then* $\vdash_{MT} t =$ “*s*” $\iff$ Π *is a proof of s.*

A sequent tree term t, when corresponding to a proof Π in OT, can be proved equal to the quotation mark name of the theorem proved by Π and viceversa. From theorem 1 and lemma 1 we can thus derive the following result:

Corollary 1 (Reflection) : *Let* Π *be any sequent tree of s in OT. Let* “*s*” *be the quotation mark name of s. Then* $\vdash_{MT} T(\text{“}s\text{”}) \iff$ Π *is a proof of s.*

In other words $\vdash_{MT} T(\text{“}s\text{”}) \iff \vdash_{OT} s$. Then the reflection rules [Giunchiglia and Smaill, 1989]

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

are (a sort of) derived inference rules between theories in the multitheory system MT- OT. Notice that the procedural distinction between "being an already asserted theorem" and " being a theorem to be asserted", lost by m_{up}, is brought back by the reflection rules. In fact, R_{up} can be executed only on theorems already asserted in OT, while, viceversa, R_{down} can be executed to assert new theorems in OT. Occurrences of *T* that in the compilation down from MT to GET would be translated into IS-A-THEOREM correspond to applying Ru_P ; viceversa, the occurrences of *T* that would be compiled down into proof-add-theorem correspond to applying R_{down}. The use of reflection up and down allows us to give a declarative explanation of the interaction between reasoning in OT and reasoning in MT and, in particular, of how (and why) it is possible to assert theorems in OT as a result of deduction in MT.

5.3 Deducing in OT via reasoning in MT

In this section we show how a twff can be interpreted to prove a theorem in OT. As a prototypical example, let us consider the following twff:

$$T(\text{“}s\text{”}) \wedge Forall(\text{“}s\text{”}) \wedge Conj(alle(\text{“}s\text{”}, \text{“}x\text{”}, \text{“}x\text{”})) \wedge NoFree(\text{“}x\text{”}, ande(alle(\text{“}s\text{”}, \text{“}x\text{”}, \text{“}x\text{”}))) \rightarrow T(alli(ande(alle(\text{“}s\text{”}, \text{“}x\text{”}, \text{“}x\text{”})), \text{“}x\text{”}, \text{“}x\text{”}))$$

Let us take s as a shorthand for $\forall x(A(x) \wedge B(x))$. In this case the above twff is a theorem in MT and “represents” the proof implemented by the program **(all-distr-conj s x x)** (see section 2). Given the sequent s asserted in OT we can apply R_{up} to obtain $T(\text{“}s\text{”})$ in MT. $T(\text{“}s\text{”})$ can be used to derive

$$Forall(\text{“}s\text{”}) \wedge Conj(alle(\text{“}s\text{”}, \text{“}x\text{”}, \text{“}x\text{”})) \wedge NoFree(\text{“}x\text{”}, ande(alle(\text{“}s\text{”}, \text{“}x\text{”}, \text{“}x\text{”}))) \rightarrow T(alli(ande(alle(\text{“}s\text{”}, \text{“}x\text{”}, \text{“}x\text{”})), \text{“}x\text{”}, \text{“}x\text{”}))$$

The interpretation of $Forall(\text{“}s\text{”})$ leads to the execution of **(FORALL s)** which evaluates to **TRUE**. The same happens with all the other conjuncts. Thus, by simple propositional reasoning, it is possible to derive $T(alli(ande(alle(\text{“}s\text{”}, \text{“}x\text{”}, \text{“}x\text{”})), \text{“}x\text{”}, \text{“}x\text{”}))$.

The interpretation of $alle(\text{“}s\text{”}, \text{“}x\text{”}, \text{“}x\text{”})$ leads to the execution of **(alle-fun s x x)** which evaluates to $A(x) \wedge B(x)$. The result of the interpretation of the whole term is then $\forall x A(x)$. Reflection down can thus be used to assert the new theorem in OT:

$$R_{down}\frac{\vdash_{MT} \quad T(\text{“}\forall x A(x)\text{”})}{\vdash_{OT} \quad \forall x A(x)}$$

The code performing the above steps is implemented in GETFOL and can be run by the command REFLECT [Giunchiglia and Smaill, 1989]. *Notice that running the code compiled by the "mapping down" would give the same result as running* REFLECT. The inverse of m_{up} is to REFLECT exactly what compilation is to interpretation: execution of code generated by the inverse of m_{up} produces the same results as interpreting metalevel theorems via REFLECT.

6 Conclusions and related work

In this paper we have presented a theorem prover, GETFOL, where the underlying code has been written to

By (CON J A_i AB_i,) we mean the result of the application of the GET function CONJ to its arguments.

[8]Note that truth is tested in the standard model. It is well known that the set of wffs true in a model is larger than the set of valid wffs. On the other hand it can be proved that, the interpretation in GETFOL restricted to atomic ground wffs (the elements of *V)* returns TRUE iff the wff is valid in all models of MT and thus provable in MT.

behave as the procedural rnetalevel of the logic it implements. This approach seems a first step towards the development of systems able to modify deductively and automatically their underlying computation machinery. In fact:

(a) a logical metatheory MT can be automatically generated from the code;

(b) (some of) the theorems of MT represent object level computations;

(c) these theorems can be automatically compiled back in the system code to extend or to modify it (modification is achieved by redefining GET function symbols);

(d) these theorems can be automatically interpreted to assert object level theorems. In this case, as a side effect, we have a proven correct way to mix, *at run time,* object and rnetalevel theorem proving via the use of reflection up and down. More on this issue can be found in [Giunchiglia and Traverso, 1990] which also has a long section on the related work, in particular with [Bundy, 1988; Weyhrauch, 1980].

As far as we know, this approach is new and has never been proposed before. However, some comparisons with existing systems can nevertheless be made.

The idea of a metatheory mapped directly from the system code is somehow similar to the idea underlying the work on metafunctions [Boyer and Moore, 1981] (in the Boyer and Moore theorem prover the code *is* the metatheory). In [Boyer and Moore, 1981], user defined term-rewriting functions can be checked to verify whether they preserve the "meaning" of terms. Aside from the technical differences, a fundamental difference is that we provide a metatheory in which we can perform automatic deduction to *build* correct control strategies, while Boyer and Moore *verify* the correctness of the user defined strategies.

Besides Boyer and Moore's work [Boyer and Moore, 1981], none of the existing theorem provers, has the possibility of using the results of deduction in the metatheory to produce modifications of the underlying system code. This is, for instance, the case also in NuPrl [Constable *et* al., 1986; Howe, 1988], even if in NuPrl the synthesis of new tactics can be obtained by metatheoretic theorem proving (via the "propositions-as-types" paradigm). Analogously, metainterpreters can control the Prolog search strategy but 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.

For what concerns the issue of self-modification, the work which most closely resembles ours is Brian Smith's [Smith, 1983]. The substantial difference is that, in GETFOL, metatheoretic statements are generated by rnetalevel deduction and not by computation and that the tactics derived are provably correct. No non-theorems can be proved.

Aknowledgments

The authors thank the Mechanized Reasoning Group at IRST and the Mathematical Reasoning Group in Edinburgh, in particular Alan Bundy, David Basin, Alessandro Cimatti, Luciano Serafini, Alex Simpson and Alan Smaill. Vanni Criscuolo, Bob Kowalski, Carolyn Talcott, Frank VanHarmelem and Richard Weyhrauch are also thanked.

References

[Bowen and Kowalski, 1982] K.A. Bowen and R.A. Kowalski. Amalgamating language and meta-language in logic programming. In S. Tarlund, editor, *Logic Programming,* pages 153-173, New York, 1982. Academic Press.

[Boyer and Moore, 1981] 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.

[Bundy, 1988] A. Bundy. The Use of Explicit Plans to Guide Inductive Proofs. In R. Luck and R. Overbeek, editors, *CADE9.* Springer-Verlag, 1988. Longer version available as DAI Research Paper No. 349, Dept. of Artificial Intelligence, Edinburgh.

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

[Giunchiglia and Smaill, 1989] F. Giunchiglia and A. Smaill. Reflection in constructive and non-constructive automated reasoning. In J. Lloyd, editor, *Proc. Workshop on Meta-Programming in Logic Programming.* MIT Press, 1989. IRST Technical Report 8902-04. Also available as DAI Research Paper 375, Dept. of Artificial Intelligence, Edinburgh.

[Giunchiglia and Traverso, 1990] F. Giunchiglia and P. Traverso. Plan formation and execution in a uniform architecture of declarative metatheories. In M. Bruynooghe, editor, *Proc. Workshop on Meta-Programming in Logic.* MIT Press, 1990. Also available as IRST Technical Report 9003-12.

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

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

[Paulson, 1989] L. Paulson. The fundation of a generic theorem prover. *Journal of Automated Reasoning,* 5:363-396, 1989.

[Smith, 1983] B.C. Smith. Reflection and Sematincs in LISP. In *Proc. 11th ACM POPL,* pages 23-35, 1983.

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

[Weyhrauch, 1980] R.W. Weyhrauch. Prolegomena to a theory of Mechanized Formal Reasoning. *Artificial Intelligence. Special Issue on Non-monotonic Logic,* 13(1), 1980.