

ISTITUTO PER LA RICERCA SCIENTIFICA E TECNOLOGICA

I 38100 TRENTO – LOC. PANTÉ DI POVO – TEL. 0461–814444

TELEX 400874 ITCRST – TELEFAX 0461–810851

INTROSPECTIVE METATHEORETIC REASONING

Fausto Giunchiglia

Alessandro Cimatti

November 1992

Technical Report # 9211-21

ISTITUTO TARENTINO DI CULTURA

Introspective Metatheoretic Reasoning^{*}

Fausto Giunchiglia^{1,2} Alessandro Cimatti¹

¹ IRST, 38050 Povo, Trento, Italy

² University of Trento, Via Inama 5, 38100, Trento, Italy

Abstract. This paper describes a reasoning system, called GETFOL, able to introspect (the code implementing) its own deductive machinery, to reason deductively about it in a declarative metatheory and to produce new executable code which can then be pushed back into the underlying implementation. In this paper we discuss the general architecture of GETFOL and the problems related to its implementation.

1 The Goal

The work partially described in this paper tackles the problem of investigating criteria and techniques for the development of real introspective/reflective systems. We describe a system, called GETFOL³ [Giu92], able to introspect its own code, to reason deductively about it in a declarative metatheory and, as a result, to produce new executable code which can then be pushed back into the underlying implementation. The behaviour of this system is schematized in three steps (see figure 1). We call *Lifting* the first step, projecting computation into deduction. With lifting, the source code of the system is processed, and a formal metatheory describing it is automatically generated. Then, during the *Reasoning* step, the theorem proving capabilities of the system are used to reason about the lifted theory. This allows to deduce theorems which can be interpreted as specifications of new system functionalities. We call *Flattening* the last step, dual of lifting, projecting deduction into computation. By flattening, new pieces of executable code are automatically generated from statements proved with the reasoning step, and added to the system code. In this way the system may be extended, adding new inference procedures, and modified, substituting old versions of inference procedures with new, possibly optimized, ones.

Figure 2 shows the conceptual dependencies among the entities involved in the process of figure 1. The starting point is the object theory OT. We give a computational account of OT, i.e. we mechanize it, by writing code. By devising MT, we give a declarative characterization of OT. The lifting and flattening procedures act as a bridge between the code implementing OT and its metatheory MT. We call the process described in figure 1, *introspective metatheoretic reasoning* (IMR). *Introspective* because the system is able to formalize and reason

^{*} This work has been done at IRST as part of the MAIA project.

³ GETFOL is a reimplement/extension of the FOL system [Wey80, GW91]. GETFOL has, with minor variations, all the functionalities of FOL plus extensions, some of which described here, to allow for metatheoretic theorem proving.

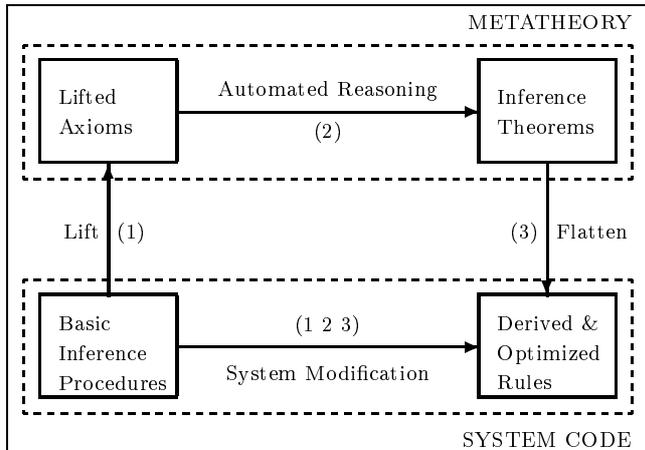


Fig. 1. The lifting-reasoning-flattening cycle.

about (parts of) its own code. *Metatheoretic* because we require that MT be a *metatheory* of the theory OT mechanized by the system code. Notice that reasoning in a formal metatheory gives the right perspective for describing inference mechanisms. In general one would like to prove that a new inference procedure is derived from, admissible or consistent with the already existing ones. This is the kind of results that metatheoretic reasoning provides (see for instance [BM81, GS88, Pau89]). However MT is different from any other metatheory defined in the past. Not only does it describe the object level logic, but it also takes into account the computations implementing it. Its theorems can be interpreted in terms of both the object level logic and its mechanization. To point out this fact, we say that MT is a *metatheory of a mechanized object theory*.

Some observations. The work closest in spirit to ours is Smith's [Smi83]. In Smith's reflective 3-Lisp, the flow of computation can be *inspected* by reifying the status of the interpreter in explicit data structures, and *modified* by suitably updating these data structures. In both his and our approaches, what Smith calls an *embedded account* of the system, i.e. a (partial) description of the system in the system itself, can be automatically generated and the system's behaviour can be modified. The basic difference is in the way the system is modified. In 3-Lisp these changes can not be declaratively reasoned about but only performed procedurally by computation.

On the other hand, our architecture is based on a sharp distinction between *computation* and *deduction*. There are many reasons underlying this choice. Epistemologically, we believe that deduction and computation are two fundamentally different phenomena. Technically, using logic gives a semantics which allows us to make and prove correctness statements. Implementationally, we can use and integrate different techniques for solving the different aspects of the problem. For instance theorem proving techniques can be used for the automation of the reasoning step, while techniques for the synthesis, optimization and verification

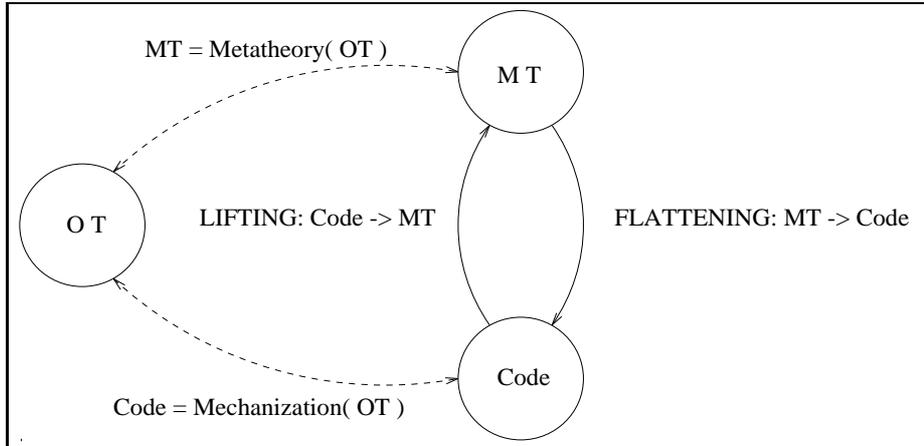


Fig. 2. The conceptual schema of the system

of the correctness of programs can be used for the implementation of the lifting and flattening processes. Furthermore, from a cognitive science viewpoint, formal deduction seems more suited than computation for the modeling of reasoning (but see [JL83]).

Though distinguished, computation and deduction are related. This is a well known fact in the mathematical logic literature. It is in fact possible to express deduction in terms of computation (e.g. in the λ -calculus) and viceversa deduction (e.g. in Peano arithmetics) may represent computation. A formal definition of the lifting and flattening processes, together with a proof of their completeness and correctness, could be seen as a re-statement of some of the results in the mathematical logic literature. However, these similarities are more superficial than it might look at a first sight. The results in the mathematical logic literature allow us to connect in a general way (to the extent that it is possible) deduction with computation in idealized programming languages (e.g. functional languages). In our work we have connected deduction in a formal metatheory (see section 2), particularly suited for representing search strategies [GT92], and the code of a real running system, GETFOL, with more than one MB of source code. Establishing this connection has required facing a lot of problems which are irrelevant from the point of view of mathematical logic and which are due to the fact that we are dealing with a real system. These problems and our proposed solutions are informally (and partially) described in the following of this paper. Thus, in section 2 we describe the structure of the metatheory MT, and various ways of reasoning about and extending inference mechanisms. In section 3 we discuss the problem of writing code which can be lifted. In section 4 we focus on the problems related to lifting and flattening the code of a real system, showing how MT can be automatically generated. Finally, in section 5 we draw some conclusions.

2 MT: a metatheory of a mechanized object theory

$\frac{A \wedge B}{A} \wedge\mathbf{E}$	$\frac{A(y)}{\forall x.A(x)} \forall\mathbf{I}$	$\frac{\forall x.A(x)}{A(t)} \forall\mathbf{E}$
---	---	---

Fig. 3. Some inference rules

The metatheory MT described in this section is a minor extension of the metatheory defined and formally studied in [GT91, GT92, GT94]. In this section we summarize the ideas described in those papers to the extent that it is needed for the goals of this paper.

We suppose that OT uses a first order classical sequent logic. By *sequent* we mean a pair (\star, A) , where A is a formula and \star a set of formulas. The inference rules are a sequent presentation of Prawitz' natural deduction calculus [Pra65]. They allow introduction and elimination in the post-sequent A . Some of the rules are shown in figure 3. They are one of the two conjunction elimination rules, and the rules for introduction and elimination of the universal quantifier. MT contains all the "standard" information about OT. That is, its language must have names for all the objects of OT (constants, variables, sequents, ...). It must have axioms that say, for instance, that x is a variable (i.e. $Var("x")$, where " x " is the name of x), that a sequent s is a conjunction (i.e. $Conj("s")$) and so on. This construction is routinary and it is not reported here. More interestingly, each (piece of code implementing an) inference rule of OT is represented in MT by a function symbol. The (subroutines implementing the) inference rules listed above are represented in MT by the function symbols *alli*, *alle* and *ande*. Their behaviour is described by the axioms of figure 4. T represents provability for (the mechanization of) sequents. \mathcal{A}_{ande} states that, if (the subroutine for) and elimination is applied to any (data structure representing a) provable sequent x whose formula is a conjunction, the (data structure representing the) result $ande(x)$ is also a provable sequent. The other axioms have a similar interpretation. The predicates Var , $Term$ and $Forall$ represent the (code computing the) properties of being an individual variable, a term and a sequent with a universally quantified formula. $NoFree$ represents the (code computing the) restriction on the generalized variable in the rule of forall introduction.

$(\mathcal{A}_{ande}) : \forall x.(T(x) \wedge Conj(x) \supset T(ande(x)))$
$(\mathcal{A}_{alli}) : \forall x y z.(T(x) \wedge Var(y) \wedge Var(z) \wedge NoFree(z, x) \supset T(alli(x, y, z)))$
$(\mathcal{A}_{alle}) : \forall x y.(T(x) \wedge Term(y) \wedge Forall(x) \supset T(alle(x, y)))$

Fig. 4. Axiomatization of primitive inference rules

The axioms in figure 4 allow only for reasoning about correct proofs, i.e. compositions of applicable inference steps. However, while trying to prove a theorem, rarely the user (or the system itself) has a detailed proof schema in mind, and inference rules may be tried without knowing whether they are applicable or not. In the code of GETFOL possible failures are taken into account by suitable subroutines, called *tactics*. Tactics call the basic inference rules when applicability

conditions are satisfied, otherwise return a particular data structure representing failure. In order to implement the cycle of figure 1, tactics, and therefore failure, must be represented in MT. Failure is represented by the individual constant *fail*. The basic concept of being a failure is expressed by the predicate *Fail* defined as

$$\forall x. Fail(x) \leftrightarrow x = fail$$

The axiom

$$(\mathcal{A}_{nottfail}) : \forall x. \neg(Fail(x) \wedge T(x))$$

states that *T* and *Fail* are disjoint. Being a proof step (either successful or failing) is expressed by the predicate *Tac*, defined by the axiom

$$(\mathcal{A}_{Tac}) : \forall x. Tac(x) \leftrightarrow (Fail(x) \vee T(x))$$

However, this is not enough. The metalevel representation of a tactic must have a structural similarity (very close to a one-to-one mapping) with the tactic itself. This makes the lifting, theorem proving and flattening of new tactics easier and more natural to understand and to perform. Tactics are basically programs and, as such, make extensive use of conditional constructs (and their derivatives). We have therefore extended the first order language with conditional term constructors (i.e. **trmif** *wff* **then** *term* **else** *term*), whose meaning is given by the inference rules for elimination and introduction of figure 5 (the elimination rule for *A* is not listed, being very similar to that for $\neg A$). The resulting theory is a conservative extension of first order logic. Figure 6 shows how the tactics implementing the inference rules of figure 3 are represented in MT.

$\frac{\begin{array}{c} [A] \\ \vdots \\ B(t_1) \end{array} \quad \begin{array}{c} [\neg A] \\ \vdots \\ B(t_2) \end{array}}{B(\mathbf{trmif} A \mathbf{then} t_1 \mathbf{else} t_2)} \text{trmif}I \quad \frac{\neg A \quad B(\mathbf{trmif} A \mathbf{then} t_1 \mathbf{else} t_2)}{B(t_2)} \text{trmif}E_{\neg}$

Fig. 5. Rules for conditional terms

$(\mathcal{A}_{andetac}) : \forall x. andetac(x) = \mathbf{trmif} T(x) \wedge Conj(x) \mathbf{then} ande(x) \mathbf{else} fail$
$(\mathcal{A}_{allitac}) : \forall x y z. allitac(x, y, z) = \mathbf{trmif} T(x) \wedge Var(y) \wedge Var(z) \wedge NoFree(z, x) \mathbf{then} alli(x, y, z) \mathbf{else} fail$
$(\mathcal{A}_{alletac}) : \forall x y. alletac(x, y) = \mathbf{trmif} T(x) \wedge Term(y) \wedge Forall(x) \mathbf{then} alle(x, y) \mathbf{else} fail$

Fig. 6. Definition of primitive tactics

Notice that tactics implement the total version of inference rules by returning an explicit failure. Failures are used for this particular purpose. Therefore, it

may not be the case that inference rules return a failure when the applicability conditions are not satisfied. This fact is formalized in MT by

$$(\mathcal{A}_{allinofail}) : \forall x y z. \neg all_i(x, y, z) = fail$$

and allows to keep reasoning about correct inference and general inference completely separated.

$\forall x y. alleandallitac(x, y) = \mathbf{trmif} \begin{array}{l} T(x) \wedge Forall(x) \wedge Var(y) \wedge \\ Conj(alle(x, y)) \wedge NoFree(y, ande(alle(x, y))) \\ \mathbf{then} \ all_i(ande(alle(x, y)), y, y) \\ \mathbf{else} \ fail \end{array}$
--

Fig. 7. The definition of *alleandallitac*

The ultimate goal of reasoning in MT about an inference procedure is to prove the corresponding *admissibility statement*, and then to flatten it down as system code. For inference rules, such statements have the form of axioms of figure 4, i.e.

$$\forall \underline{x}. Applicability(\underline{x}) \supset T(rule(\underline{x}))$$

The admissibility statements for the corresponding tactics have the form

$$\forall \underline{x}. Tac(ruletac(\underline{x}))$$

In MT the admissibility statements for an inference rule and for the corresponding tactic can be derived from each other.

In MT and extensions of MT it is possible to synthesize/optimize tactics. In the following we will try to give the flavour of how this can be done via two examples. As an example of possible reasoning in MT, let us consider the class of *finite compositions* of inference rules. Arbitrary compositions of function symbols can be proved to satisfy *Tac* in MT. Consider the rule which is a composition of a forall elimination, a conjunction elimination and finally a forall introduction. From the admissibility statement for *allitac*, we can prove in MT $\forall x y. Tac(allitac(andetac(alletac(x, y)), y, y))$, stating that the composition is a correct inference rule. From the point of view of the logic provably equal terms are completely indistinguishable. However, the composition of tactics as above generates very redundant and inefficient code. An equivalent term defining the very same inference steps and corresponding to more *optimized* code, can be deduced by composing the axioms of figure 4:

$$\forall x y. T(x) \wedge Forall(x) \wedge Var(y) \wedge Conj(alle(x, y)) \wedge NoFree(y, ande(alle(x, y))) \supset T(all_i(ande(alle(x, y)), y, y))$$

The corresponding tactic is defined as:

$$\forall x y. alleandallitac(x, y) = \mathbf{trmif} \begin{array}{l} T(x) \wedge Forall(x) \wedge Var(y) \wedge \\ Conj(alle(x, y)) \wedge NoFree(y, ande(alle(x, y))) \\ \mathbf{then} \ all_i(ande(alle(x, y)), y, y) \\ \mathbf{else} \ fail \end{array}$$

alleandeallitac is optimized with respect to the composition of primitive tactics, as the applicability is expressed in a simplified way. For instance, the theoremhood tests performed by *andetac* and *allitac* are eliminated as they are implied by the other applicability conditions. Notice that this framework allows for incremental reasoning about inference procedures: first a basic version of inference rule can be considered, (e.g. by composing tactics) and then it can be more and more optimized by deducing further properties.

$\forall x.uniclosetac(x) = \mathbf{trmif} \ T(x)$ $\quad \mathbf{then} \ \mathbf{let} \ (var \ . \ get\text{-}free(x))$ $\quad \quad \mathbf{in} \ \mathbf{trmif} \ Var(var)$ $\quad \quad \quad \mathbf{then} \ \mathit{uniclosetac}(\mathit{allitac}(x, var, var))$ $\quad \quad \quad \mathbf{else} \ x$ $\quad \mathbf{else} \ fail$

Fig. 8. Definition of *uniclosetac*

In (extensions of) MT it is also possible to reason about *admissible* inference procedures, i.e. rules which do not enlarge the provability relation of OT, but which can not be expressed as a finite composition of primitive inference steps. Being able to reason about this kind of inference rules gives the system the capability of extending itself with non trivial functionalities: typical examples are a tautology decider or a normalization procedure. (Our treatment of admissible rules is similar to Boyer and Moore's work on metafunctions [BM81].) As a simple example, consider the inference rule performing the universal closure of an arbitrary sequent. This rule succeeds provided that none of the free variables of the formula occurs free in the dependencies. This rule can be represented in MT by the function symbol *uniclosetac*, defined by the formula of figure 8. The definition exploits another conservative extension of first order language, the environment term constructor, i.e. **let** (*var . term*) **in** *term* (which can be defined similarly to what done for conditional terms). The corresponding admissibility statement is the formula $\forall x.Tac(uniclosetac(x))$. Its proof requires reasoning by induction with the axiom schema of figure 9. The intuition underlying the proof is to perform an induction on the structure of well formed formulas (wffs). At every step the selected free variable is proved to be quantified by the application of *allitac*.

$\mathcal{A}_{ind} : \forall x \ y. (Wff(x) \wedge Wff(y) \supset (\Phi(x) \wedge \Phi(y) \supset \Phi(mkor(x, y)))) \wedge \dots \wedge$ $\quad \forall x \ y. (Var(x) \wedge Wff(y) \supset (\Phi(y) \supset \Phi(mkforall(x, y)))) \wedge \dots \supset$ $\quad \forall x. (Wff(x) \supset \Phi(x))$

Fig. 9. The structural induction axiom schema for wffs

3 Writing liftable code

The goal is to produce code which, once lifted, will generate MT (see figure 2). However, this is not a trivial consequence of the fact that the code im-

```
(DEFLAM alliprf (X Y Z)
 (IF (AND (THEOREM X) (VAR Y) (VAR Z) (NOFREE Z X))
  (proof-add-theorem (alli X Y Z))
  (print-error-message)))
```

Fig. 10. Unliftable code for forall introduction

plements the deductive machinery of OT. Writing liftable code is not a simple operation of implementation. It is not enough to satisfy the usual software engineering requirements (e.g. bug-free, understandable). The code must preserve a form of structural similarity with the entity being represented, in this case OT. Every data structure, every step of computation, the system structure and the abstraction levels are determined in terms of the concepts that are intended to be relevant. To emphasize this point we say that we do *representation theory using programs as representational tools*. We call this way of writing code, *mechanization*, and distinguish it from simple implementation.

Consider the code implementing the simple inference rule of forall introduction, shown in figure 10 (the programming language is HGKM [GC89], a SCHEME-like language with first order semantics, used as the implementation language of GETFOL). THEOREM evaluates to TRUE if the argument is (a data structure representing) a provable sequent, VAR evaluates to true if the argument is a variable, alli builds a sequent and proof-add-theorem adds it to the proof. Its reading is very natural: if the inference rule is applicable then build the (data structure representing) the resulting sequent and store it in the current proof, otherwise report an error message. Everyone would agree that this is well written code. But the correspondence with the description in MT of $\forall I$, is only partially preserved. For instance, whilst *allitac* in MT is a function from sequents to sequents, alliprf has a side effect on the system (either adds a sequent to the proof or prints an error message) and does not return a value.

The work on mechanizing GETFOL has required different rewritings of the code, during which we have devised a general schema for the development of liftable code (see figure 11). We call *state* the information stored in the system. The state is of different kinds, and is justified by different reasons. For instance, the axioms of the object theory are stored because of their logical relevance. A counter for the automatic generation of different names for skolem functions is stored for user interface purposes. In decision procedures, global variables are used to avoid explicit argument passing and optimize performances. We call *logical state* (LS) the part of the state containing logical information, *physical state* (PS) the remaining state. This distinction separates the information which is relevant for lifting (i.e. LS) from the one which must be suitably “hidden” by identifying in the code a liftable abstraction level. The system code is then separated in operations which are functional on the state, the *computation machinery* (CM), and operations which change the state, the *update machinery* (UM). Notice that this classification is completely general and independent of

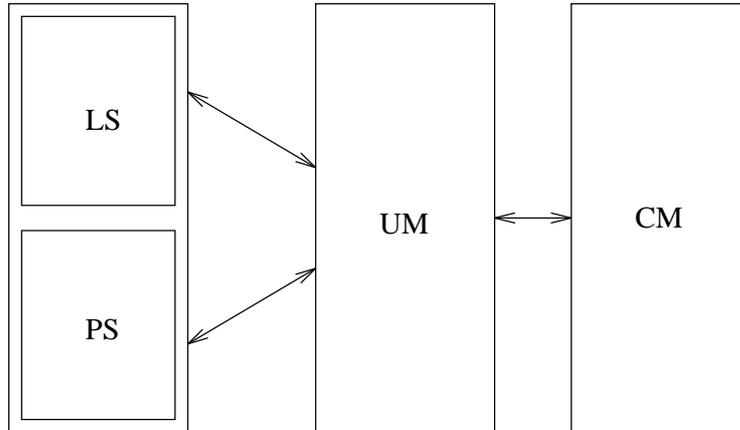


Fig. 11. The structure of the system code.

the application.

The implementation of forall introduction according to this schema is given in figure 12. `allitac` and `alli` are CM primitives. `allitac` implements the forall introduction tactic, while `alli` implements $\forall I$ assuming that the arguments satisfy the applicability conditions. `alli` and `allitac` are functions over a fixed state, as their execution does not add a theorem to the current proof nor does it print an error message. `proof-add-theorem` is the UM primitive “hiding” part of LS, i.e. the current proof, to which it adds the given theorem. `print-error-message` “hides” part of the physical state, i.e. the output channel, on which it prints an error message. These operations are called by the UM primitive `alliprf`, according to the result of `allitac`. Compare the two implementations of $\forall I$ in figure 10 and in figure 12. The computations described in the two cases are very similar, undistinguishable from many points of view. However, the abstraction levels in the “mechanized” solution are sharply identified, the functions are separated from the actions on the state. Even more important, notice the correspondence between the subroutines `alli` and `allitac` with the axiomatization in MT of the rule of forall introduction (figure 6).

4 Lifting and flattening code

As already pointed out in section 1, the relation between computation and deduction has been widely studied in mathematical logic. In computer science, programming languages are given formal account to allow formal reasoning for program synthesis, optimization and verification. Independently of the features of the language being described, i.e. functional [BM79] or imperative [MW87a, MW87b], all these approaches are based on a *uniform* mapping. In most cases, the computing subroutines are immersed into the logic with a mapping which is basically one-to-one. Our approach takes a different perspective. Lifting and

```

;;; UPDATE MACHINERY
(DEFNAM alliprf (X Y Z)
  (maybe-proof-add-theorem (allitac X Y Z)))

(DEFNAM maybe-proof-add-theorem (X)
  (IF (FAIL? X)
    (print-error-message X) ;;; UPDATE THE PHYSICAL STATE
    (proof-add-theorem X))) ;;; UPDATE THE LOGICAL STATE

;;; COMPUTATION MACHINERY
(DEFNAM allitac (X Y Z)
  (IF (AND (THEOREM X) (VAR Y) (VAR Z) (NOFREE Z X))
    (alli X Y Z)
    fail))

```

Fig. 12. Lifiable code for forall introduction

flattening *are not uniform*: different parts of the code are treated in different ways.

In this paper, for lack of space, we do not consider flattening. We rely on the intuition that the flattening step of the schema in figure 1 is the inverse of the lifting. Given a definitional axiom, it generates the corresponding one-to-one CM definition. For instance, flattening the definition of *uniclose* given in figure 8 gives the CM function *uniclosetac* (figure 13). Statements about the system state, once proved in MT, are flattened onto UM primitives. For instance, the admissibility statement $\forall x.Tac(uniclosetac(x))$ is flattened in the UM primitive *unicloseprf*.

Lifting is defined according to the classification of the code discussed in previous section. Basically, for each of LS, CM and UM there is a corresponding lifting procedure generating an appropriate subset of axioms of MT. In this paper we do not consider lifting of LS. The intuition is that LS contains information related to the objects of OT (e.g. the language, the axioms), and lifting LS gives their metatheoretic description, e.g. the formula *Var*(“*x*”) for the (data structure implementing the) variable *x* of OT. We focus here only on the harder cases of lifting CM and lifting UM.

4.1 Lifting CM

The code in CM is a collection of functions defined using the functional subpart of HGKM. Therefore lifting CM can exploit the usual techniques for reasoning about functional programs: function definitions are immersed via a one-to-one mapping into the logic of MT and become definitional axioms. For instance, lifting the function definition of *allitac* (figure 12) gives the axiom $\mathcal{A}_{allitac}$ (figure 6).

```

(DEFLAM unicloseprf (x)
  (maybe-proof-add-theorem (uniclosetac x)))

(DEFLAM uniclosetac (seq)
  (IF (NOT (THEOREM seq)) fail
    (LET ((v (get-free seq)))
      (IF (VAR v)
        (uniclosetac (allitac seq v v))
        v))))

```

Fig. 13. Flattening *uniclose*

In principle, it would be possible to lift all the function definitions of the system going down to the basic primitives, e.g. `CAR`, `CDR`, `CONS`. This is for instance what Boyer and Moore do; higher levels of functional abstraction are added incrementally during the theorem proving activity [BM79]. Their “bottom-up” approach is motivated by the fact that their goal is to prove the termination and the correctness (with respect to a certain specification) of user defined functions. However, we are interested in developing systems which reason selectively about portions of their underlying implementation code. The idea is to keep MT as partial as possible still maintaining all the needed information. This in order to make the process of self-extension focused and feasible in practice. Therefore, we take a slightly different approach. First of all, we want reasoning to be *local*: for instance, we do not want to consider the system deciders when reasoning about inference rules. Furthermore, reasoning should be *at the right level of abstraction*. For instance, in order to reason about tactics, we are not interested in the internal structure of inference rules, and we take `alli` to be a primitive object, i.e. as if it were a black box. We call *abstract machine* the collection of all and only the primitive objects involved in our reasoning (e.g. `alli`). Notice that the choice of what code we lift, and at which level of abstraction we describe it, strongly depends on the goals: for instance, taking inference rules as primitive objects is particularly suited for synthesizing new tactics.

The hard problem in lifting CM is to lift the suitable formalization for the abstract machine. Indeed, this can not be described with the one-to-one lifting of its subroutine definitions: this would imply reasoning at a different (lower) level of abstraction. The properties of an abstract machine must be lifted without analyzing its internal structure. General criteria for axiomatizing abstract machines have been defined. A first issue is the characterization in MT of partial functions (e.g. `alli`). A partial function is associated with a total version (e.g. `allitac`) which returns the special data structure `fail` when the value is not defined. In order to avoid confusion, partial functions never return `fail`. This general property has been exploited to define a lifting mechanism returning the corresponding axiom: in the case of `alli` and `allitac`, the resulting axiom is $\mathcal{A}_{allinofail}$ (see page 6). A second issue is the lifting of code imple-

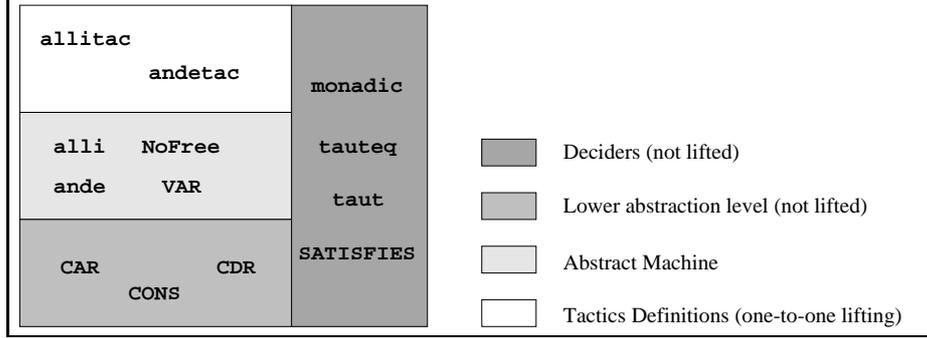


Fig. 14. Lifting the structure of CM

menting inductively defined data structures. Consider the code in figure 15. It is a (simplified) version of the implementation of wffs. In this code it is possible to identify constructors (`mkor`), recognizers (`DISJ`) and selectors (`lfor`, `rtor`). Starting from such considerations, we can lift the axioms defining the mutual relations of constructors and selectors, e.g.

$$\mathcal{A}_{lformkor} : \forall x y. lfor(mkor(x, y)) = x$$

The concrete implementation of this code is not shown, as lifting is independent of it. The recursively defined predicate `WFF` “glues together” the various objects into an inductively defined data type. From each of its clauses, we can lift the type information for constructors, e.g.

$$\mathcal{A}_{typemkforall} : \forall x y. Var(x) \wedge Wff(y) \supset Wff(mkforall(x, y))$$

In the case of inductively defined data, it is also possible to lift a principle of structural induction. For `WFF` the result is the axiom \mathcal{A}_{ind} used in the admissibility proof of `uniclose` (see figure 9).

4.2 Lifting UM

Lifting the theory of inference is somehow analogous to lifting the theory of wffs. Indeed, the set of provable sequents of OT can be seen analogously to wffs, i.e. an inductively defined set, whose constructors are the inference rules. (One minor difference is that for wffs we only consider the partial functions, (e.g. the computation of `lfor` may return a random value or even raise an exception if the argument is not a `DISJ`); in the case of inference we consider both (partial) inference rules and the corresponding total versions with `fail`. Under this interpretation, axiom \mathcal{A}_{ande} is the dual of the typing axiom $\mathcal{A}_{typemkforall}$.) A possible solution could be therefore to lift $\mathcal{A}_{typemkforall}$ from CM, as done for wffs, given a recursive definition of the provability predicate (dual of `WFF`). To do this, however, we would also need the recognizers for different inference steps (dual of `UNIQUANT`). The problem is that theoremhood is actually a property which is not decidable,

```

(DEFLAM mkor (wff1 wff2) ...)
(DEFLAM lfor (wff) ...)
(DEFLAM rtor (wff) ...)
(DEFLAM DISJ (wff) ...)
...
(DEFLAM mkforall (var wff) ...)
(DEFLAM bvarof (wff) ...)
(DEFLAM matrix (wff) ...)
(DEFLAM UNIQUANT (wff) ...)
...
(DEFLAM WFF (wff)
  (IF (DISJ wff) (AND (WFF (lfor wff)) (WFF (rtor wff)))
    ...
    (IF (UNIQUANT wff) (AND (WFF (matrix wff)) (VAR (bvarof wff)))
      ...
      FALSE)))

```

Fig. 15. The code implementing the data type WFF

and we would like to avoid using not recursive recognizers. One (partial) solution could be to axiomatize proofhood, rather than theoremhood. This is somehow similar to the (type-theoretical/algebraic) approach of [BC91, CAB⁺86]. In this approach inference rules, rather than being functions on sequents, are functions on whole proofs. One problem is that, potentially, objects have to be recomputed from scratch any time they are introduced in the system. This is acceptable for wffs, whose data structures are indeed generated at parsing time. For sequents, though, building the corresponding proof may become a very hard task.

Lifting UM is based on a different idea. In GETFOL, the state is used to save partial computations: this allows us to reuse objects which have already been computed. Sequents, once proved, are asserted as theorems in the part of LS implementing the current proof (by `proof-add-theorem`). Verifying theoremhood (cfr. `THEOREM` in figure 12) amounts to searching in LS. Therefore the current proof in LS can be seen as an approximation of the non-recursive set of all the provable sequents. Lifting UM is based on the idea that, if an UM primitive adds the result of a computation to the state approximating a certain set, then the result also belongs to the set, i.e. the approximation is sound. In the particular case of inference, `alliprf` adds the result of `allitac` to the state approximating the set of (successful and failing) inference steps (this state is hidden by `maybe-proof-add-theorem`). Being *Tac* the description in MT of this set involved, by lifting `alliprf` we get the admissibility statement for *allitac*, i.e. $\forall x y z. Tac(allitac(x, y, z))$. As shown in section 2, for our purposes this is equivalent to the axiom \mathcal{A}_{all} . Following this approach, it is possible to lift from the UM all the necessary axioms. In particular, to lift the induction principle

$$\forall \underline{x}. \Phi(ruletac_1(\underline{x})) \wedge \dots \wedge \forall \underline{x}. \Phi(ruletac_n(\underline{x})) \supset (\forall x. Tac(x) \supset \Phi(x))$$

it is sufficient to notice that the *ruletac_i* are the only inference rules in the system.

5 Conclusions

In this paper we have presented the architecture and the issues related to the development of an *introspective metatheoretic* reasoning system, able to *lift* its own code into a declarative metatheory, *reason* about it and *flatten* down the resulting theorems into new system code. The theoretical foundations of this work (e.g. the formal definition of lifting and flattening, the proof of their full symmetry and correctness) are discussed in the companion paper [GC92].

References

- [BC91] 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.
- [BM79] R.S. Boyer and J.S. Moore. *A Computational Logic*. Academic Press, 1979. ACM monograph series.
- [BM81] 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.
- [CAB⁺86] R.L. Constable, S.F. Allen, H.M. Bromley, et al. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice Hall, 1986.
- [GC89] F. Giunchiglia and A. Cimatti. HGKM Manual - a revised version. Technical Report 8906-22, IRST, Trento, Italy, 1989.
- [GC92] F. Giunchiglia and A. Cimatti. Introspective Metatheoretic Theorem Proving. Technical Report 9211-22, IRST, Trento, Italy, 1992. Upgraded and extended version forthcoming.
- [Giu92] F. Giunchiglia. The GETFOL Manual - GETFOL version 1. Technical Report 92-0010, DIST - University of Genova, Genoa, Italy, 1992.
- [GS88] F. Giunchiglia and A. Smaill. Reflection in constructive and non-constructive automated reasoning. In H. Abramson and M. H. Rogers, editors, *Proc. of META-88, Workshop on Metaprogramming in Logic*, pages 123–145. MIT Press, 1988. Also IRST-Technical Report 8902-04 and DAI Research Paper 375, University of Edinburgh.
- [GT91] 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.
- [GT92] F. Giunchiglia and P. Traverso. A Metatheory of a Mechanized Object Theory. Technical Report 9211-24, IRST, Trento, Italy, 1992. Submitted for publication to: Journal of Artificial Intelligence.
- [GT94] F. Giunchiglia and P. Traverso. Program Tactics and Logic Tactics. In *Proceedings 5th Intl. Conference on Logic Programming and Automated Reasoning (LPAR'94)*, Kiev, Ukraine, July 16-21, 1994. Also IRST-Technical

- Report 9301-01, IRST, Trento, Italy. Presented at the Third International Symposium on Artificial Intelligence and Mathematics, Fort Lauderdale, Florida, January 1994.
- [GW91] F. Giunchiglia and R.W. Weyhrauch. FOL User Manual - FOL version 2. Manual 9109-08, IRST, Trento, Italy, 1991. Also DIST Technical Report 91-0006, DIST, University of Genova.
- [JL83] P. N. Johnson-Laird. *Mental Models*. Cambridge University Press, 1983.
- [MW87a] Z. Manna and R. Waldinger. The Deductive Synthesis of Imperative LISP Programs. In *Sixth National Conference on Artificial Intelligence*. AAAI, 1987.
- [MW87b] Z. Manna and R. Waldinger. How to clear a block: A Theory of Plans. Technical Report STAN-CS-87-1141, Department of Computer Science, Stanford University, 1987.
- [Pau89] L. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5:363–396, 1989.
- [Pra65] D. Prawitz. *Natural Deduction - A proof theoretical study*. Almqvist and Wiksell, Stockholm, 1965.
- [Smi83] B.C. Smith. Reflection and Semantics in LISP. In *Proc. 11th ACM POPL*, pages 23–35, 1983.
- [Wey80] R.W. Weyhrauch. Prolegomena to a Theory of Mechanized Formal Reasoning. *Artificial Intelligence*, 13(1):133–176, 1980.