[YS92] A. Yonezawa and B.C. Smith, editors. *Proceedings of the IMSA'92 International Workshop on Reflection and Meta-level Architecture*, 1992.

[JA92] S. Jagannathan and G. Agha. A Reflective Model of Inheritance. In *The Sixth European Conference on Object-Oriented Programming*, number to appear in LNCS, 1992.

[MW87] Z. Manna and R. Waldinger. The Deductive Synthesis of Imperative LISP Programs. In *Sixth National Conference on Artificial Intelligence*. AAAI, 1987.

[Pau79] L. Paulson. Tactics and Tacticals in Cambridge LCF. Technical Report 39, Computer Laboratory, University of Cambridge, 1979.

[Pau89] L. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5:363–396, 1989.

[Smi83] B.C. Smith. Reflection and Semantics in LISP. In *Proc. 11th ACM POPL*, pages 23–35, 1983.

[SWY91] S.Matsuoka, T. Watanabe, and A. Yonezawa. Hybrid Group Reflective Architecture for Object Oriented Concurrent Reflective Programming. *Lecture Notes in Computer Science*, 512:231–250, 1991.

[Tal85] C. Talcott. *The essence of RUM: theory of the intensional and extensional aspects of LISP-type computation.* PhD thesis, Department of Computer Science, Stanford University, 1985. Also report No. STAN-CS-85-1060.

[Wey80] R.W. Weyhrauch. Prolegomena to a Theory of Mechanized Formal Reasoning. *Artif. Intell.*, 13(1):133–176, 1980.

[WT85] R.W. Weyhrauch and C. Talcott. `HGKM`: a Simple Implementation. `FOL` working paper 4, November 1985.

[WY88] T. Watanabe and A. Yonezawa. Reflection in an Object-Oriented Concurrent Language. In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 306–316. ACM Press, 1988.

[WY90] T. Watanabe and A. Yonezawa. An Actor Based Metalevel Architecture for Group-wide Reflection. In *Proceedings of the REX School/Workshop on Foundations of Object Oriented Languages*, Lecture Notes in Computer Science, May 1990.

[Yon91] A. Yonezawa. A Reflective Object Oriented Concurrent Language. *Lecture Notes in Computer Science*, 441:254–256, 1991.

[Giu92] F. Giunchiglia. The `GETFOL` Manual - `GETFOL` version 1. Technical Report 9204-01, DIST - University of Genova, Genoa, Italy, 1992. Forthcoming IRST-Technical Report.

[GMMW77] 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.

[GMW79] 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.

[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.

[GT93] F. Giunchiglia and P. Traverso. Program Tactics and Logic Tactics. Technical Report 9301-01, IRST, Trento, Italy, 1993.

[GW91] F. Giunchiglia and R.W. Weyhrauch. `FOL` User Manual - `FOL` version 2. Manual 9109-08, IRST, Trento, Italy, 1991. Also MRG-DIST Technical Report 9107-02, DIST, University of Genova.

[HL85] P.M. Hill and J.W. Lloyd. The Gödel Programming Language. Technical Report CSTR 92-27, University of Bristol, Dept. Computer Science, 1985.

[HL89] P. M. Hill and J. W. Lloyd. Analysis of meta-programs. In J. Lloyd, editor, *Proc. Workshop on Meta-Programming in Logic Programming*. MIT Press, 1989.

[How88] D. J. Howe. Computational metatheory in Nuprl. In R. Lusk and R. Overbeek, editors, *CADE9*, 1988.

[IMWY91] Y. Ichisugi, S. Matsuoka, T. Watanabe, and A. Yonezawa. An Object Oriented Concurrent Reflective Architecture for Distributed Computing Environments. In *Proceedings of the 29th Allerton Conference on Communication, Control and Computing*, October 1991.

References

[BK82] 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.

[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.

[BM90] R.S. Boyer and J.S. More. 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.

[Bun88] 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.

[BvHHS90] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence no. 449.

[BW85] K.A. Bowen and T. Weiberhg. A Meta-level Extension of Prolog. In *IEEE Symposium on Logic Programming*, pages 669–675, Boston, 1985.

[CAB$^+$86] R.L. Constable, S.F. Allen, H.M. Bromley, et al. *Implementing Mathematics with the NuPRL Proof Development System.* Prentice Hall, 1986.

[GA93] F. Giunchiglia and A. Armando. A Conceptual Architecture for Introspective Systems. Forthcoming IRST-Technical Report, 1993.

[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 Reasoning. Technical Report 9211-21, IRST, Trento, Italy, 1992.

5 Conclusions

In this paper we have presented how computational reflective capabilities and mechanized logical deduction can be integrated. We have explained how this integration is achieved in `GETFOL`, a theorem prover where a logical metatheory MT describes both the logical properties of the object theory OT and the code implementing deduction in OT. This seems a promising step towards systems that are able to modify deductively and automatically their underlying computation machinery. In fact:

1. The code implementing the logical metatheory MT constitutes the embedded account of the system, *i.e.* the computational description of the system code.

2. Self introspection can be automated by means of a general purpose *lifting* process that, given the system code and data structures, automatically generates the embedded account.

3. Self modification can be performed correctly by means of a general purpose *flattening* process that, given theorems generated by the theorem prover, maps them into either new or new versions of programs

4. The mechanized MT and the lifting and flattening processes allow for a flexible control over computational reflection, *i.e.* the object level computation system and its metalevel representation can be connected with different, application dependent control mechanisms.

Acknowledgements

The work presented in this paper owes a lot to all the members of the Mechanized Reasoning Groups in Trento and Genoa. The authors would like to thank in particular Fausto Giunchiglia that has supervised and worked at the `GETFOL` project since 1989: without him the research described in this paper could have never been done. Other members of the Mechanized Reasoning Group in Trento and in Genoa are working on related issues. This work has been also strongly influenced by the work done by Richard Weyhrauch on `FOL`. Alessandro Armando, Paolo Pecchiari, Carolyn Talcott and Richard Weyhrauch have provided useful feedback on various aspects of the work described in this paper.

taken this work as a starting point to define the correct and complete logical metatheory MT, to write the suitable system code and relate it to MT, and to define and implement lifting and flattening thus providing computational reflection via logical deduction.

Other deductive reasoning systems have been proposed that provide logical reflection, but that do not provide computational reflection, *i.e.* that do not have the possibility of using the results of deduction to produce modifications of the underlying system code. Notice that this is the case also of deductive systems that provide a logic that can be mapped to programs and to computation, like for instance [CAB$^+$86, How88], [BK82, HL89] and [BM81]. In fact, in these systems, logical statements can be executed as programs, but these are not the programs used to implement the system itself. They cannot thus use the results of deduction in the metatheory to produce modifications of the underlying system code. An exception is the work (in progress) with the Acl2 theorem prover [BM90]: its goal is to prove the correctness of the theorem prover within the theorem prover itself. This work should raise a lot of very interesting issues in building correct self reflective systems.

In the programming language community a lot of work has addressed the issue of self-reflection (see for instance [YS92, Smi83, JA92, WY88, Yon91, IMWY91, SWY91, WY90]). The main difference with `GETFOL` is the way in which computational reflection is performed. In `GETFOL` computation is affected by using logical deduction in MT. This opens up the possibility of reasoning deductively about computation and of performing safe self-extension and self-modification. Moreover, computational reflection in `GETFOL` has been achieved in two different steps that are related, but are kept sharply distinct on purpose. In the first step (described in section 2), we have provided the necessary link between the system code and its embedded account. In the second step (described in section 3) we have defined lifting and flattening, that is how this link can be used to build the embedded account automatically and to affect computation. Keeping these two steps distinct implies a peculiar feature: *when and how lifting and flattening are to be applied is left unspecified*, *i.e.* the connection is not restricted to any "hardwired" control strategy. In `GETFOL`, proving a statement in MT allows for but does not necessarily imply flattening the corresponding code. Similarly, writing a new program or modifying an existing program, allows for but does not necessarily imply lifting its metalevel specifications. The computational system and its embedded account may evolve in their own, independently within the correctness constraints deriving from the other. This fact leaves open the possibility to causally connect the object level system and its metalevel representation with different control mechanisms according to the application requirements. This approach is very different from what happens in other reflective systems where the control over the connection is hardwired and precompiled in the system.

the reasoning process feasible, we want it to be *local* and *at the right level of abstraction*. For instance, in order to reason about tactics, we do not want MT to consider the system deciders, nor are we interested in the internal structure of inference rules (the choice of what code we lift, and at which level of abstraction we describe it, strongly depends on our goals). Once the right level of abstraction is determined, reasoning can exploit the usual techniques: `HGKM` 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 `mptac` (figure 4) gives axiom 4. Of course, one could describe `mp` with a one-to-one lifting of its subroutine definition, assuming that formulas are primitive objects; in this case axiom 3 would be derivable from the axioms lifted from the code implementing wffs. However, this would imply reasoning at a different (lower) level of abstraction. Therefore a uniform mapping is not satisfactory for our purposes. The primitive entities of MT, such as mp, should be described as if they were black boxes. Lifting the corresponding axioms is indeed the difficult task: indeed, the relevant code must be lifted without analyzing the internal structure, *i.e.* its definition. In this case lifting is rather based on the role that the code plays within the system. For instance, in order to lift the functional code implementing an inductively defined data type, such as wffs, lifting can exploit the role of the different functions, *e.g.* constructors, selectors and recognizers. The task becomes more difficult once we consider non-functional code: the corresponding state has to be taken into account in lifting. This is the case for lifting of inference rules: formulas, once proved, are asserted (by `proof-add-theorem`) as theorems in the state implementing the current proof; verifying theoremhood (see `THEOREM` in figure 4) amounts to searching in the corresponding data structure. Lifting the code of inference is based on the idea that the proof can be seen as an approximation of the (non-recursive) set of all the provable formulas (represented in MT by T): if the code adds a data structure to the state approximating the set of theorems, then this data represents a theorem, *i.e.* the approximation is sound. In the particular case of the code for modus ponens, `mpprf` adds the result of `mp`, under the applicability conditions determined by `mptac`, to the state approximating the set of theorems (by calling `proof-add-line`). Under this interpretation the lifting procedure generates axiom 3.

4 Related Work

The major influence on this work comes from the work done in the `FOL` system [Wey80], one of the first deductive systems with self-reflective capabilities. The idea of reflection rules has been first proposed in `FOL`. `FOL` had an implementation of reflection down. The idea of enforcing a correspondence between symbols of the language of a logical metatheory to `HGKM` code was originally proposed and implemented in `FOL`. The `GETFOL` project has of lifting is in its generality, as the technology applied is general purpose, *i.e.* independent of the fact that we are lifting the code of a theorem prover. In general this is an important feature as the interesting characteristics of the embedded account of a refelctive system can depend on the particular introspection to be performed. The peculiar feature of flattening as a self modification process is the starting point, *i.e.* logical MT statements, which are *de facto* formal specifications for system modifications/extensions. The advantage is that in this way the synthesized code is guaranteed to satisfy the specifications, and therefore the process of self extension can be proved to be correct.

The definition of the lifting and flattening relations are by no means trivial, not even assuming the conditions on MT and on the structure of the system code discussed in previous section. Lifting and flattening are in fact mapping between a computational system (the code implementing `GETFOL`) and a deductive system (MT). In the past such relation has been widely studied. It is a well known fact in the mathematical logic literature that it is possible to express deduction in terms of computation (*e.g.* in the λ-calculus) and vice versa deduction (*e.g.* in Peano arithmetics) may represent computation; in computer science, programming languages are given formal account to allow formal reasoning for program synthesis, optimization and verification. However these results are obtained in rather idealized settings. Here we take a different perspective. Establishing the connection between MT and the code mechanizing OT has required facing a lot of problems, which arise just because we are dealing with a real full blown system. Independently of the features of the language being described, *i.e.* functional [BM79] or imperative [MW87], all the approaches to mapping between computation and deduction 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. This is of course a good property, that would make the implementation of lifting nearly trivial. However, in particular in the case of languages with imperative constructs and state, these approaches seem quite far apart from being applicable to our problem. The code of `GETFOL` amounts to more than one MB of source, and uses a lot of state: reasoning in the theory describing it in a one-to-one fashion would simply be unfeasible in practice. Therefore lifting and flattening *are not uniform*: different parts of the code are treated in different ways. The result is a one-to-one correspondence only for a particular class of programs; however, this allows for the expression of computations we are interested in reasoning about.

In the following we focus on the description of lifting; we rely on the intuition that flattening can be seen as the inverse process. The generation of the language of MT and the description of the data structures in OT is somewhat standard: for instance, the MT axiom $T(``\forall x.P(x)")$ is the lifting for the (data structure implementing the) axiom $\forall x.P(x)$ of OT. Let us consider the problem of lifting code. We are interested in developing a system which reasons about its underlying implementation code. In order to maintain

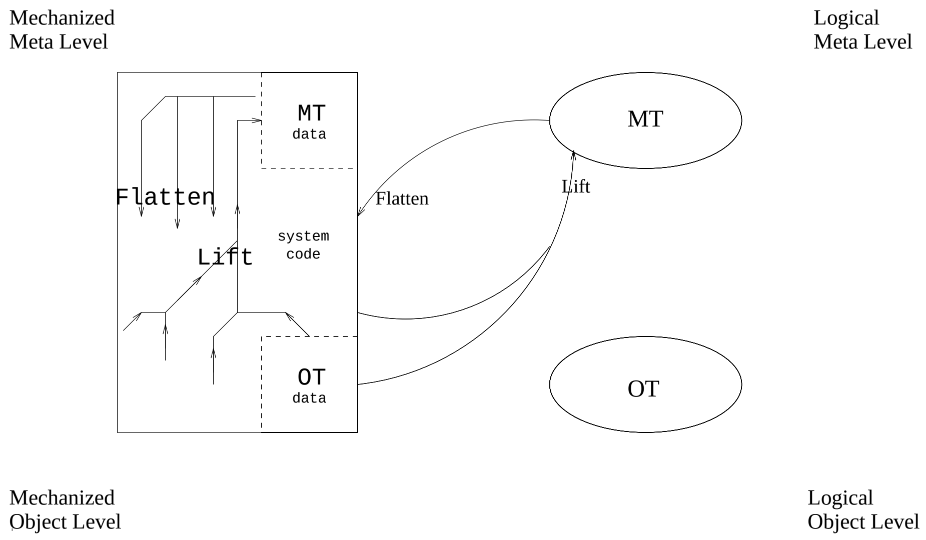

Figure 5: Lifting and flattening in `GETFOL`

determined.

The connection between MT and the system code is exploited in the reverse direction with the *flattening* process. Flattening is basically a process of executable code synthesis. MT theorems can be interpreted as descriptions of pieces of executable (system) code satisfying certain specifications. Flattening, given such specification, (automatically) generates the corresponding executable code.

The arrows `Lift` and `Flatten`, in figure 5, depict the implementation of lifting and flattening, which have effect on the system data structures and code. `Lift` has the actual effect of generating the data mechanizing MT, *i.e.* to build the embedded account of the system and its causal connection to the system code. `Flatten` has the effect of generating new parts of the system code starting from the theorems of MT; in this way the system can perform self-extension, when the added code implements new functionalities, and self-modification, when the added code is a substitute for old functionalities.

It is then clear in which sense these functionalities allow for computational reflection in `GETFOL`: the computations (implementing deduction) in MT are actually about the computation mechanisms of the system itself, and the flattening process, exploiting the causal connection between the mechanized MT and the system code, performs system modification and extension.

Lifting is the basis for the automatization of the self introspection process. Notice that the generation of the embedded account is vital to any reflective system. The main advantage

```
(DEFLAM mpprf (X Y)
 (maybe-proof-add-theorem (mptac X Y)))

(DEFLAM maybe-proof-add-theorem (X)
 (IF (FAIL? X)
  (print-error-message)
  (proof-add-theorem X)))

(DEFLAM mptac (X Y)
 (IF (AND (THEOREM X) (THEOREM Y) (HP X Y))
  (mp X Y)
  fail))
```

Figure 4: The mechanization of modus ponens in `GETFOL`

latter takes into account the way inference rules may not be applicable. Other primitives implement operations which have no direct correspondence in OT (*e.g.* handling of the input output channel), but which are vital to the implementation of a full blown reasoning system.

Compare now the two versions of `mpprf` in figure 3 and in figure 4. 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, and the functions are separated from the actions on the state; this does not happen in the other implementation, where the different levels are collapsed together. Furthermore, notice how the mechanization of modus ponens of figure 4 yields the natural correspondence between the definition of `mptac` and axiom 4.

3 Computational Reflection in `GETFOL`

The connection between MT and the system code described in previous section allows for the implementation of the functionalities schematized in figure 5. *Lifting* is a theory-building process, whose result is the (automated) generation of the description of the system code, *i.e.* MT. With lifting, the system code and data structures are analyzed, and as a result the necessary language and axioms are automatically generated. Furthermore, the connections between the language of MT and the corresponding data structures are

```
(DEFLAM mpprf (X Y)
 (IF (AND (THEOREM X) (THEOREM Y) (HP X Y))
  (proof-add-theorem (mp X Y))
  (print-error-message)))
```

Figure 3: A possible implementation of modus ponens

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 [GC92].

Consider the code implementing the rule of modus ponens, shown in figure 3. `THEOREM` evaluates to `TRUE` if the argument is (a data structure asserted as) a theorem, `HP` evaluates to true if the first argument is an hypothesis of the second, `mp` builds (the data structure representing) the conclusion of the inference, 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 formula and store it in the current proof, otherwise report an error message". Everyone would agree that this is well written code. However, it is hard to see how this code could be related to the representation of modus ponens in MT. For instance, whilst *mptac* in MT describes modus ponens as a binary function mapping wffs to wffs, `mpprf` implements modus ponens with a side effect on the system (either adds a theorem 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 general criteria for the development of code with the required properties. An example of this way of writing code is the mechanization of modus ponens in `GETFOL`, given in figure 4. `mp` is the function that, given two formulas that satisfy the applicability conditions, returns the result of the inference. `mptac` is the modus ponens tactic, which maps two data structures in the conclusion of the inference if the preconditions are satisfied, otherwise returns `fail`, the data structure representing failure. `FAIL?` tests whether its argument is `fail`. `proof-add-theorem` adds its argument to the proof, while `print-error-message` prints an error message. The general idea is to enforce a sharp identification of the levels of abstraction and of the state of the system involved in the representation of OT, and to separate it from other parts of the code which have no strict representational meaning. `mp` and `mptac` take into account two different aspects of inference in OT: the former captures the notion of "correct" inference, while the

In the mechanized OT, in addition to building proofs, it is possible to define *tactics*[3], *i.e.* programs which implement search strategies for proofs. Tactics specify which inference rules to apply, in which order and under which conditions, and how to recover when the chosen rule is not applicable; they are HGKM complex programs containing conditionals, iterations and failure generation and trapping constructs. Tactics are a very relevant part of the code mechanizing deduction in GETFOL. MT provides for representing and reasoning about tactics. In order to show how this is done, we consider the simple tactics corresponding to the application of a single inference rule. These tactics are mechanized so that they allow for failure trapping and composition into more complex tactics. The MT axiom describing the tactic corresponding to the rule of modus ponens is:

$$\forall xy.mptac(x, y) = \mathbf{trmif}\ T(x) \wedge T(y) \wedge Hp(x, y)\ \mathbf{then}\ mp(x, y)\ \mathbf{else}\ fail \qquad (4)$$

The individual constant $fail$ represents failure in the applicability of inference steps. **trmif** is the MT constructor for conditional terms, namely terms whose denotation depends on the interpretation of the condition wff. The intuition is that when the applicability conditions of the inference rule are satisfied, mp is called, otherwise a failure is generated. This axiom reflects the implementation code. However, in MT it is possible to reason about the computation in a declarative way. Failure trapping is simply equivalent to proving the equality of the corresponding tactic with $fail$. Inference rules to introduce and eliminate conditional expressions [Giu92] allow for reasoning deductively about conditionals. This expressiveness of MT allows for using effectively the theorem proving capabilities of the system to reason about a relevant portion of the code implementing GETFOL and settles the basis to extend and modify it deductively.

2.2 The system code

Our goal is to produce code implementing OT that can also be described by a logical metatheory of OT. Notice that this is not a trivial consequence of the fact that the code implements the deductive machinery of OT. Writing such a code is not a simple operation of implementation. It is not enough to satisfy the usual software engineering requirements (*e.g.* bug-free, understandable). It is aslo necessary that the code preserves 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 represented. To emphasize this point

[3]Historically, tactics have been mostly developed in the programming language ML [GMMW77] and used in LCF and its descendants [GMW79, Pau79, Pau89, CAB$^+$86]. There actually exist technical differencies between GETFOL tactics and LCF tactics, that we do not discuss here since out of the goal of this paper.

connect the embedded account to the system code and data structures.

The existence of a causally connected embedded account is a first step towards computational reflection. However, achieving the necessary relation between MT and the code as described above is by no means trivial. It requires that MT and the code be designed to satisfy some fundamental conditions. In sections 2.1 we discuss necessary requirements in the definition of MT; in section 2.2 we discuss some issues in writing the system code.

2.1 The metatheory

There are actually several different ways to set up the signature and the axioms of MT such that MT is a metatheory of the formal OT. Consider the object level inference rule of modus ponens:

$$\frac{A \quad A \supset B}{B}\, mp \tag{1}$$

At the metalevel, modus ponens could be represented by the axiom:

$$T(``A") \wedge T(``A \supset B") \supset T(``B") \tag{2}$$

"A", "$A \supset B$" and "B" denote the object level wffs A, $A \supset B$ and B, respectively. Axiom 2 states that if A and $A \supset B$ are theorems, then B is a theorem. Axioms of this form may suffix to provide a correct and complete representation of OT provability.

However, as we want MT to take into account the *mechanization* of OT, this choice is not satisfactory. Axiom 2 has no correspondence with the structure of the system code: it states the provability of the conclusion of the rule given the premises, independently of the computations implementing the application of the inference rule. The mechanization of modus ponens in `GETFOL` is based on the `HGKM` predicate `HP`, that tests the precondition of the rule and returns `TRUE` iff its arguments are (data structures representing) wffs of the form $A \supset B$ and A, and on the `HGKM` function `mp` that, given (the data structures corresponding to) the premises A and $A \supset B$, returns (the data structure corresponding to) the conclusion of the rule B. The MT axiom

$$\forall x\ y.T(x) \wedge T(y) \wedge Hp(x,y) \supset T(mp(x,y)) \tag{3}$$

describes modus ponens preserving a structural similarity with the system code mechanizing it. The binary function symbol mp corresponds to `mp`, and the binary predicate symbol Hp corresponds to `HP`. The other inference rules of OT are represented in MT according to the criteria shown in this simple example.

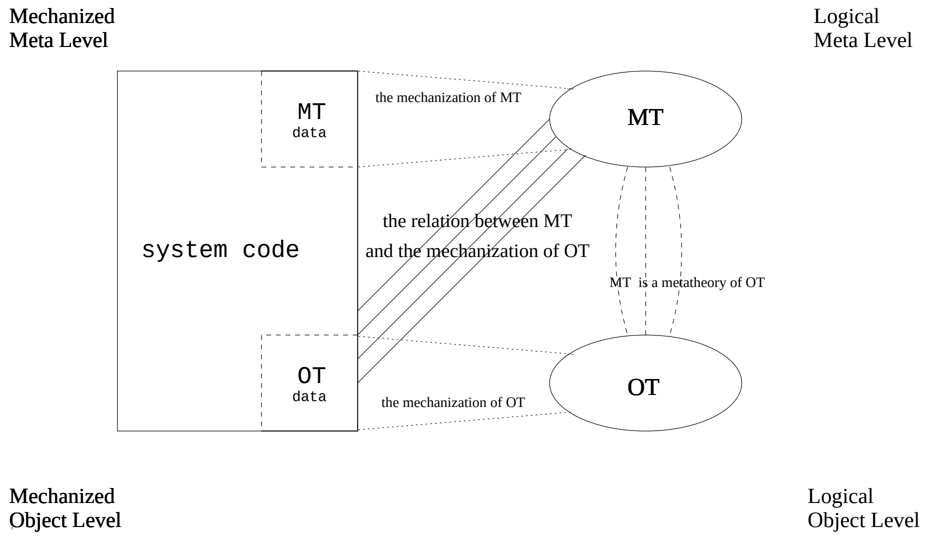

Figure 2: The link between MT and the code in `GETFOL`

rem provers, *e.g.* [CAB$^+$86, How88, HL85, BW85, BvHHS90]. The logical metatheories implemented there describe the object theory, but have no relation with the code and the data implementing the object theory. MT has instead a distinguishing characteristic that makes it different from any other logical metatheory proposed so far. Not only does MT describe the logical theory OT, but it also takes into account the fact that OT is mechanized in `GETFOL`. It also describes the way in which deduction in OT is actually performed by running code. This results in a well defined relation between the language of MT and (a relevant subset of) `OT data` and the system code: deduction in MT corresponds to computation in the code implementing OT. The symbols in $\mathcal{ML}$ are in a one-to-one correspondence with `HGKM` [2] computational entities in the mechanization of OT. For any individual constant in $\mathcal{ML}$ (*e.g.* "$\forall x.A(x)$") there is a corresponding data structure (*e.g.* the s-expression `(forall x (P x))`) representing an object of OT (*e.g.* the formula $\forall x.P(x)$). For any function or predicate symbol (f and P) in the language of MT there is a corresponding `HGKM` function (`f` and `P`) in the code implementing OT. This relation between the logical metatheory MT and the mechanized object theory is depicted in figure 2. As a consequence of this relation, the `GETFOL` implementation of MT, *i.e.* `MT data`, plays the role of *embedded account* in `GETFOL`: indeed, MT describes actual data and programs of `GETFOL`, and `MT data` encodes this (partial) representation of the system in the system itself. Furthermore, being `MT data` mechanized within the system itself, it is possible to implement the link between MT and the code, *i.e.* to causally

[2]`HGKM` is the implementation language of `GETFOL` [Tal85, WT85, GC89].

2 The relation between MT and the code

Consider first a generic first order logical theory, OT, defined as the 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. Given OT, we define the first order logical theory MT $= < \mathcal{ML}, \mathcal{MA}, \mathcal{MR} >$, to be its metatheory. We fix a *naming relation* between MT and the objects of OT, *i.e.* $\mathcal{ML}$ contains names of the objects (*e.g.* terms, wffs, axioms) of OT. For instance, the constants "$\forall x.A(x)$" and "x" are the names of the OT wff $\forall x.A(x)$ and of the OT variable x, respectively. The $\mathcal{ML}$ statement $T(``A")$ expresses the provability in OT of the formula $A$. $\mathcal{MA}$ axiomatizes the OT provability relation so that MT is correct and complete *w.r.t.* OT, *i.e.* $T(``A")$ is provable in MT iff A is provable in OT. This fact provides a theoretical justification for the *reflection rules* R_{up} and R_{dw} between MT and OT depicted in figure 1. The premise of reflection up is the OT wff A, and its conclusion the MT wff $T(``A")$; reflection down works in the opposite way, *i.e.* maps $T(``A")$ in MT to A in OT.

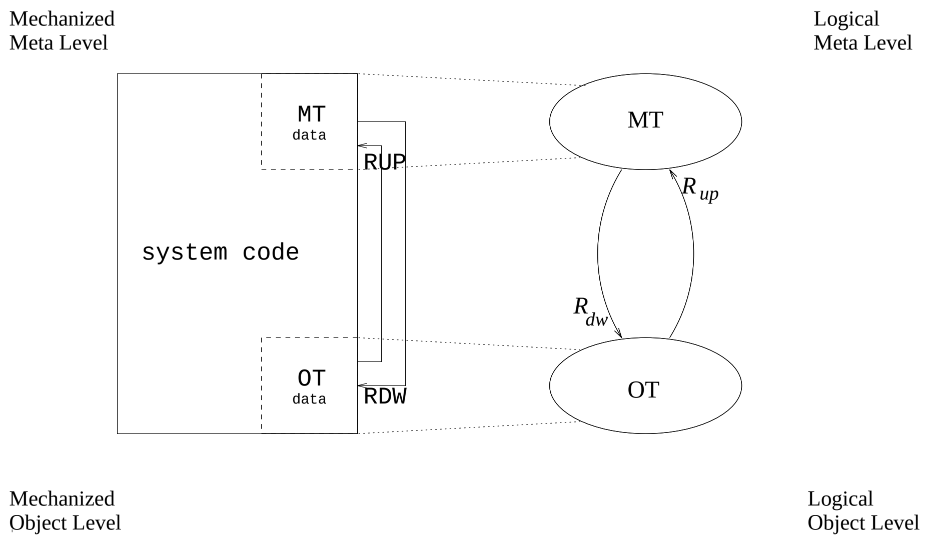

Figure 1: `GETFOL` implements logical reflection

`GETFOL` allows for the implementation of MT and OT as distinct contexts, shown in figure 1 as `OT data` and `MT data`. Each context contains data structures implementing the language, the set of axioms and the set of inference rules of each logical theory. R_{up} and R_{dw} are mechanized by the system commands `RUP` and `RDOWN` (see figure 1), that assert a theorem in `MT data` given a theorem in `OT data` as input, and vice versa.

MT, as described so far, plays a role similar to metalevel representations in other theo-

1. There exists a relation between MT and the code implementing OT. The same expressions of MT referring to the logical properties of OT are also in a well defined relation with the code implementing logical deduction in OT. Not only does MT reason about (logical properties of) OT, but also reasons about (computational properties of) the system code implementing logical deduction.

2. The relation between MT and the code is exploited by *lifting* and *flattening*, two general and domain independent procedures, that map MT into the code and vice versa. Lifting, given in input the code implementing the object theory, generates the data structures implementing MT. Flattening, given in input data structures implementing MT, generates system code.

As a consequence of the first fact, the implementation of MT is indeed the computational description of the system code, or, according to Smith's terminology, the *embedded account* of the system [Smi83]. As a consequence of the second fact, lifting builds the embedded account and its causal connection to the system code, while flattening affects systems computation by extending/modifying the underlying computational strategies. Therefore, the machinery performing deduction in MT is the basis for the machinery that is used to perform computational reflection. This achieves our final goal, *i.e.* computational reflection via mechanized logical deduction.

`GETFOL` is developed within a global project that has gone on at the Mechanized Reasoning Group of IRST since 1989. The ideas presented in this paper have been developed starting from the results of existing research within this project that involves the authors and other members of the group. The aim of this paper is to interpret and understand this research under a different perspective, thus opening the way to use computational reflection (via mechanized logical deduction) as an important tool to achieve the project's goals. Therefore, many technical details are omitted here. The interested reader is deferred to other papers: [GT91, GT92, GT93] describe in detail MT and how MT takes into account the code implementing deduction in OT, while [GC92] describes the lifting and flattening processes, the `GETFOL` code and how it must be written to allow for a mapping with MT; [GA93] describes a conceptual architecture to integrate logic and computation in `GETFOL`.

The paper is structured as follows. Section 2 describes how MT, the logical object theory and the code implementing it are related. Section 3 describes how this relation is used to achieve computational reflection (*i.e.* lifting and flattening). In sections 4 we discuss the relation of our work with some related systems. In section 5 we give some conclusions.

1 Introduction and motivations

Reflective and metatheoretic reasoning are well known techniques applied in automated deduction (see for instance [Wey80], [Bun88], [BM81], [CAB$^+$86, How88], [BK82, HL89]). Proof checkers and theorem provers have been proposed that implement logical metatheories, *i.e.* logical theories whose language has expressions referring to other logical (object) theories or even to themselves. Logical metatheories can reason deductively about and affect logical deduction within other theories. We call this ability, *logical reflection*. However, these systems do not have the ability to affect their own computation. Their data and programs do not represent and compute about data and programs of the system itself. In a word, they do not provide *computational reflection*.

We are interested in the integration of computational reflection and deduction in automated reasoning systems. Our goal is to present how a system for automated deduction can be given computational reflection by using the very same reasoning capabilities of the system itself. Such a system has the ability to affect its own computation mechanism, and the machinery used to affect its own computation is based on the machinery implementing logical deduction. We call this feature *computational reflection via mechanized logical deduction*. "Computational reflection", since the theorem prover can access and modify its own computation. "Via mechanized logical deduction", since the basis for this computational reflection is inference within the mechanized logical framework.

There are some main motivations for this work. On the one hand, there is the intrinsic interest in how self-reflective computation and mechanized deduction can be related, combined and integrated. On the other hand, providing computational reflection within a deductive reasoning system presents significant advantages from the practical point of view. First, the theorem prover can inspect, extend and modify its own underlying theorem proving strategies automatically. Second, mechanized logical deduction can be used to reason about the ways these strategies can be extended and modified. Since logic gives a semantics which allows us to make and prove correctness statements, this approach opens up the possibility of building systems that are able to perform correct and safe self-extension and self-modification.

In this paper we show how computational reflection is achieved via mechanized logical deduction within the deductive reasoning system `GETFOL` [Giu92] [1]. Similarly to other theorem provers, `GETFOL` implements logical reflection using a metatheory, called MT, describing the logical object theory (OT). In addition, two main distinguishing features allow us to achieve the desired properties.

[1] `GETFOL` has been developed on top of a reimplementation [GW91] of the `FOL` system [Wey80].

Computational Reflection via Mechanized Logical Deduction[*]

Alessandro Cimatti and Paolo Traverso
IRST - Istituto per la Ricerca Scientifica e Tecnologica
38050 Povo, Trento, Italy
cx@irst.it, leaf@irst.it

Abstract

In this paper, we show how a system for automated deduction can be given computational reflection, *i.e.* can compute about and affect its own computation mechanism, by using the very same machinery implementing logical deduction. This feature, that we call *computational reflection via mechanized logical deduction*, provides both theoretical and practical advantages. First, the theorem prover can inspect, extend and modify its own underlying theorem proving strategies automatically. Second, mechanized logical deduction can be used to reason about the ways these strategies can be extended and modified and to prove correctness statements. This opens up the possibility of building systems that are able to perform correct and safe reflective self-extension and self-modification.

[*]This work has been conducted as part of MAIA, the integrated AI project under development at IRST.

Istituto per la Ricerca Scientifica e Tecnologica
I 38100 Trento – Loc. Panté di Povo – tel. 0461–314444
Telex 400874 ITCRST – Telefax 0461–302040

Computational Reflection via Mechanized Logical Deduction

A. Cimatti, P. Traverso

December 1993
Technical Report # 9312-02

Istituto Trentino di Cultura