Automating Meta-Theory Creation and System Extension

David Basin* Fausto Giunchiglia† Paolo Traverso‡

Abstract

In this paper we describe a first experiment with a new approach for building theorem provers that can formalize themselves, reason about themselves, and safely extend themselves with new inference procedures. Within the GETFOL system we have built a pair of functions that operate between the system's implementation and a theory about this implementation. The first function *lifts* the actual inference rules to axioms that comprise a theory of GETFOL's inference capabilities. This allows us to turn the prover upon itself whereby we may formally reason about its inference rules and derive new rules. The second function *flattens* new rules back into the underlying system. This provides a novel means of safe system self-extension and an efficient way of executing derived rules.

1 Introduction

Theorem proving systems are generally viewed as static systems or black boxes. One cares not how the underlying rules are implemented, but only that they mimic the inference rules of some desired logic and that they only allow valid formulas to be proven. The focus of this paper is a new approach to automating meta-theory creation and extension in which the black box is opened and the actual implementation of a theorem prover (GETFOL [GT90a]) provides the basis for a formal theory about the prover, a *theory of implementation*.

There are many possible theories of implementation; for example, theories of inference, theories of syntax, and theories of binding. Each corresponds to the axiomatization of a specific part of the underlying system. The previous examples correspond respectively to axiomatizing the procedures that specify the prover's inference rules, term structure, and variable binding. Which theory is formalized determines the kind of theorems one can prove about the system.

In this paper, we formalize a theory of inference where axioms are provided to reason about the system's inference rules. Within the GETFOL system we have built a pair of functions that operate between the system's implementation of inference rules and a

*DAI, U. of Edinburgh, EH1 1HN, Scotland. (basin@aipna.ed.ac.uk)
†IRST and DIST, U. of Genoa, Via Opera Pia 11A, Genova, Italy. (fausto%irst@uunet.uu.net)
‡IRST, Povo, 38100 Trento, Italy. (leaf%irst@uunet.uu.net)

theory about this implementation. The first function *lifts* the actual inference rules to axioms that comprise a theory of GETFOL's inference capabilities. The second function *flattens* derived rules back into the underlying system. Within the formalized theory we can turn the prover upon itself, formally reason about its inference rules, and derive new rules. By this means, we may derive any inference rule that corresponds to some finite application of primitive rules. These derived rules are similar to the class of inference rules derivable in Paulson's Isabelle system [Pau89]. And as in Isabelle, we have a guarantee of correctness.

This process of lifting, proving, and flattening, is illustrated in Figure 1. The lifting step

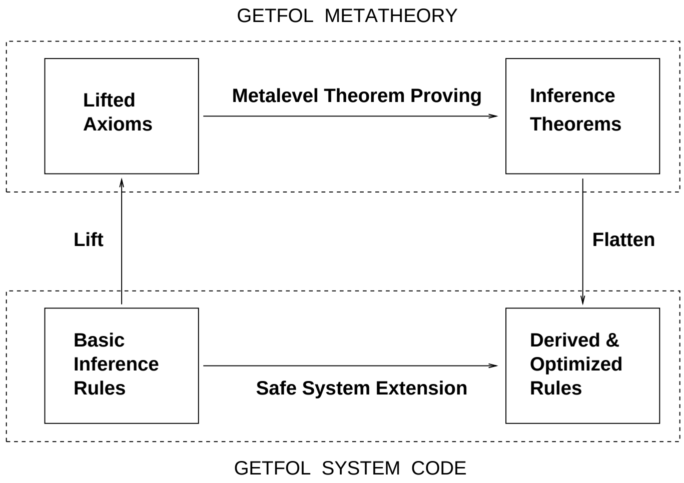

Figure 1: Lifting and Flattening Cycle

builds the metatheory from scratch. Deduction in the metatheory generates theorems about the inference rules. Compilation down expands the system, adding new code that implements the new derived rules. This cycle may be iterated; successive iterations may use rules derived on previous rounds either as inference rules or as axioms to derive further inference rules in the current theory of inference.

This approach provides a foundation for self-reasoning and self-extension. In addition, it can provide considerable efficiency advantages, both intellectual and computational. First, new derived inference rules can be used to shorten subsequent proofs. Second, derived rules may be applied without the overhead of reflection; their flattened images are executed as compiled code like the rest of the system. Third, logical manipulation at the theory level corresponds to program transformation at the system level; hence, the system may be safely optimized via logical manipulation. For example, logical manipulations that shorten the length of a rule or replace functions with simpler counterparts will generally compile to a faster executing rule. Finally, our rules explicitly represent the proof tree data-structure and this allows us to perform optimization in the meta-theory via partial evaluation. We shall later provide examples of such optimizations.

Our paper is organized as follows. Section 2 provides background on GETFOL and indicates the central role of this system in creating a theory of inference. Section 3 describes lifting, the process of automatically generating a theory of the implementation from the implementation. Section 4 shows how new inference rules can be derived in this theory. Section 5 explains flattening, the dual of lifting, whereby new rules are compiled down into the underlying Lisp system and indicates why the entire lifting/deriving/flattening process preserves theorem-hood in GETFOL. Section 6 demonstrates how logical manipulation at the theory level corresponds to function transformation and optimization at the system level. In Section 7, some related work is considered, and in the final section, we draw conclusions and indicate directions for further research.

2 The Role of GETFOL

Our work occurs in the context of GETFOL [GT90a] a reimplementation and extension of the FOL system [Wey80]. GETFOL was carefully coded[1] to facilitate the automatic axiomatization of parts of the system. This would not be possible for an ordinary implementation; we believe that systems of the kind we propose require that

1. the actual inference procedures are coded in a manner whereby these procedures can be given a declarative axiomatic reading (*e.g.*, a clausal manner); and
2. these procedures directly construct and manipulate the actual proof data-structure.

The first point is necessary if we are to be able to generate axioms automatically from the implementation itself. The second is required as our objective is to reason about and modify the GETFOL implementation and control strategy without having to reason about or modify a layer of meta-interpretation. In the following section we indicate that the inference functions of GETFOL do meet these criteria.

These system design requirements, so central to our approach, are non-trivial to achieve simultaneously, although individually each is easily achieved. For example, if GETFOL had been implemented in a pure logic programming language (not Prolog!), the first objective could be satisfied but not the second. On the other hand, one may write inference rules that directly construct and modify proof trees, but considerable care must be taken if these routines are to be automatically axiomatizable. Hence, much care was taken to design GETFOL to not only have proper extensional properties (*i.e.*, the inference rules compute the right proof data structures) but to have these intensional properties as well. After this, the definition and implementation of the lifting and flattening functions was relatively simple.

3 Lifting

GETFOL allows the definition of multiple distinct first-order theories, each a Natural Deduction calculus [Pra65]. In this section we describe how we automatically create a

[1] GETFOL was written in a Lisp-like functional language GET. The details of the implementation and the semantics of GET are provided in [GT90c] but are are not relevant here. We will subsequently speak as if Lisp were the implementation language.

```
<C-RULE>   ::=   (DEFLAM <NAME> (<VAR> ...  <VAR>) (IF <C-PROP> <C-PROP>))
<C-PROP>   ::=   (AND <C-PROP> <C-PROP>) |
                 (<PRED-CONSTANT> <C-TERM> ...  <C-TERM>)
<C-TERM>   ::=   <VAR> | (<FUNC-CONSTANT> <C-TERM> ...  <C-TERM>)
```

Figure 2: Grammar for Lisp Subset

theory of inference corresponding to the GETFOL implementation. We proceed as follows. First, we circumscribe the subset of Lisp that liftable GETFOL procedures must be written in; it corresponds to a clausal language. All GETFOL inference rule procedures have been written in this Lisp subset. Second, we define the lifting function as a family of functions that maps programs written in the Lisp subset to axioms in a formal theory. Our description will be brief as these mappings are fairly direct. One subtlety we address is that the procedural distinction between *determining if* a formula holds and *asserting that* a formula holds under assumptions collapse declaratively when there is no notion of execution and evaluation order. Hence, these concepts must be collapsed at the theory level.[2] We conclude with an example.

Figure 2 defines our axiomatizable Lisp subset. Some explanations are in order. DEFLAM is the equivalent of Lisp's DEFUN and starts a function definition. The NAME of a function is a unique (from other function names) character string. Similarly, declared variables (members of VAR) must also be unique strings. Variable occurrences in functions are restricted to these named variables. We have not further specified the PRED-CONSTANT, and FUNC-CONSTANT syntactic categories. These may be any predicate (functions returning TRUE or FALSE) or function defined in the underlying code. Also not explicitly mentioned is that each such symbol has an associated arity and the number of arguments to which it is applied must respect its arity. Constants are considered to be 0-ary functions. Finally, although only the propositional connective AND is used in our current work, others may be easily added.

```
(DEFLAM impe-rule (fact1 fact2 )
  (IF (AND (AND (IS-FACT fact1) (IS-FACT fact2))
       (HYP-OF fact1 fact2))
            (proof-add-theorem (impe fact1 fact2))))

(DEFLAM alli-rule (fact v1 v2)
  (IF (AND (AND (IS-FACT fact) (AND (IS-VAR v1) (IS-VAR v2)))
      (NO-FREE v2 fact))
           (proof-add-theorem (alli fact v1 v2)))))
```

Figure 3: Implication Elimination and All Introduction

For example, Figure 3 contains the code that implements the following Natural Deduction inference rules.

$$\supset E \quad \frac{A \quad A \supset B}{B} \qquad\qquad \forall I \quad \frac{A(v_1)}{\forall v_2 A(v_2)} \tag{1}$$

[2]Note that this procedural/declarative distinction would be confused in a (semi-)declarative language such as Prolog where one augmented the proof database using the procedural predicate ASSERT.

Here IS-FACT is a function that evaluates to TRUE when its argument is a hypothesis in the current deduction. A *fact* in GETFOL is the analogue of a theorem under assumptions. HYP-OF evaluates to true when its second argument is a formula whose main symbol is implication and the implicand matches HYP-OF's first argument. NO-FREE checks the restriction required by Natural Deduction on $\forall I$. The function proof-add-theorem augments the current deduction with a new fact.

$$\mathcal{L}_{rule}(\texttt{DEFLAM}\ name\ (V_1, ..., V_n)\ (\texttt{IF}\ p_1\ p_2)) =$$
$$\texttt{axiom}\ name\texttt{:}\ \ \forall V_1 ... \forall V_n\ \mathcal{L}_{prop_l}(p_1) \supset \mathcal{L}_{prop_r}(p_2)$$

$$\begin{array}{lcl}
\mathcal{L}_{prop_l}(\texttt{(AND}\ p_1\ p_2\texttt{)}) & = & \mathcal{L}_{prop_l}(p_1) \wedge \mathcal{L}_{prop_l}(p_2) \\
\mathcal{L}_{prop_l}(\texttt{(IS-FACT}\ t\texttt{)}) & = & \texttt{T}(\mathcal{L}_{term}(t)) \\
\mathcal{L}_{prop_l}(\texttt{(}P\ t_1\ \ldots\ t_n\texttt{)}) & = & P(\mathcal{L}_{term}(t_1), \ldots, \mathcal{L}_{term}(t_n)) \\
 & & \textit{for } P \neq \texttt{proof-add-theorem} \\
 & & \\
\mathcal{L}_{prop_r}(\texttt{(AND}\ p_1\ p_2\texttt{)}) & = & \mathcal{L}_{prop_r}(p_1) \wedge \mathcal{L}_{prop_r}(p_2) \\
\mathcal{L}_{prop_r}(\texttt{(proof-add-theorem}\ t\texttt{)}) & = & \texttt{T}(\mathcal{L}_{term}(t)) \\
\mathcal{L}_{prop_r}(\texttt{(}P\ t_1\ \ldots\ t_n\texttt{)}) & = & P(\mathcal{L}_{term}(t_1), \ldots, \mathcal{L}_{term}(t_n)) \\
 & & \textit{for } P \neq \texttt{IS-FACT} \\
 & & \\
\mathcal{L}_{term}(v) & = & v \\
\mathcal{L}_{term}((f\ t_1 \ldots t_n)) & = & f(\mathcal{L}_{term}(t_1), \ldots, \mathcal{L}_{term}(t_n))
\end{array}$$

Figure 4: Lifting Functions

Our translation of a program in this language into first order logic is a direct encoding (in Lisp) of the functions recursively defined in Figure 4. In our presentation of these rules the expressions $name$, p_i, f, P, v, and t_i are meta-variables ranging over the syntactic categories NAME, C-PROP, FUNC-CONSTANT, PRED-CONSTANT, VAR, and C-TERM respectively. T is a distinct uninterpreted predicate symbol (*i.e.* we assume that there is no system function "T") and signifies the truth of a fact.

The one complication in lifting is that we wish to collapse the distinction between IS-FACT and proof-add-theorem at the theory level (we map them both to T) and restore the distinction upon flattening. To accomplish this we insist that IS-FACT only appears on the left hand side of the implication and proof-add-theorem on on the right. This is reasonable because in the underlying system the former would never be executed except as precondition to an inference rule and the later is a procedure which would only be executed for its effects. This slightly complicates our lifting and flattening functions though, as we must have both $\mathcal{L}_{prop_l}$ and $\mathcal{L}_{prop_r}$ functions to keep track of which "side" of the implication we are lifting.

As we are describing a theory of inference, all other function symbols and predicate symbols occurring in the lifted inference are uninterpreted; they lift to uninterpreted function and predicate symbols (of the same name) and flatten back down to themselves. It may not be obvious that one can effectively reason about these inference rules without further interpretation (*e.g.*, theories of syntax, variable binding, and the like). This concern is addressed in the next section.

```
axiom impe-rule:
  ∀f₁∀f₂(T(f₁) ∧ T(f₂) ∧ HYP-OF(f₁, f₂)) ⊃ T(impe(f₁, f₂)))
axiom alli-rule:
  ∀f∀v₁∀v₂(T(f) ∧ IS-VAR(v₁) ∧ IS-VAR(v₂) ∧ NO-FREE(v₂, f)) ⊃ T(alli(f, v₁, v₂)))
```

Figure 5: Translation of $\supset E$ and $\forall I$ Rules

Returning to our example, applying $\mathcal{L}_{rule}$ to the $\supset E$ and $\forall I$ inference rules given in Figure 3 yields the axioms given in Figure 5. Note that the actual code implementing $\mathcal{L}_{rule}$ has to generate automatically the appropriate signature (*e.g.*, sort declarations for the relevant syntactic entities). We do not describe this process as it is straightforward.

The lifting function automatically generates axioms that comprise the meta-theory of `GETFOL` inference capabilities. We will address the correctness of this procedure after our discussion of flattening in Section 5.

4 Self Reasoning

In [GT90b] it is explained how to combine axioms and prove theorems which may be used as new derived rules. To make the paper self contained, we will illustrate self reasoning with a rather simple example: proving that C follows from A, $A \supset B$, and $A \supset (B \supset C)$. The Natural Deduction proof is as follows.

$$\supset E\ \frac{\supset E\ \frac{A \quad A \supset B}{B} \quad \supset E\ \frac{A \quad A \supset (B \supset C)}{B \supset C}}{C} \tag{2}$$

We can derive this new rule within our `GETFOL` theory of inference from our axiom `impe-rule` (Figure 5) in the following way. Let a_1, a_2, and a_3 be free variables ranging over facts. Performing $\forall E$s on the `impe-rule` axiom with a_1 and a_2 and again with a_1 and a_3 yields the following formulas.

$$T(a_1) \wedge T(a_2) \wedge \texttt{HYP-OF}(a_1, a_2) \supset T(\texttt{impe}(a_1, a_2)) \tag{3}$$

$$T(a_1) \wedge T(a_3) \wedge \texttt{HYP-OF}(a_1, a_3) \supset T(\texttt{impe}(a_1, a_3)) \tag{4}$$

Instantiating `impe-rule` again, this time with $\texttt{impe}(a_1, a_2)$ and $\texttt{impe}(a_1, a_3)$ yields

$$\begin{array}{l} T(\texttt{impe}(a_1, a_2)) \wedge T(\texttt{impe}(a_1, a_3)) \wedge \texttt{HYP-OF}(\texttt{impe}(a_1, a_2), \texttt{impe}(a_1, a_3)) \\ \quad \supset T(\texttt{impe}(\texttt{impe}(a_1, a_2), \texttt{impe}(a_1, a_3))) \end{array} \tag{5}$$

Notice that the conclusions of Equations 3 and 4 match conjuncts in the hypothesis of Equation 5. Hence, a bit of manipulation will yield the formula

$$\begin{array}{l} (T(a_1) \wedge T(a_2) \wedge \texttt{HYP-OF}(a_1, a_2) \wedge T(a_1) \wedge T(a_3) \wedge \\ \quad \texttt{HYP-OF}(a_1, a_3) \wedge \texttt{HYP-OF}(\texttt{impe}(a_1, a_2), \texttt{impe}(a_1, a_3))) \\ \quad\quad \supset T(\texttt{impe}(\texttt{impe}(a_1, a_2), \texttt{impe}(a_1, a_3))) \end{array} \tag{6}$$

Finally, applying $\forall I$ completes the GETFOL derivation

$$\begin{aligned}&\forall f_1 \forall f_2 \forall f_3((T(f_1) \wedge T(f_2) \wedge \texttt{HYP-OF}(f_1, f_2) \wedge T(f_1) \wedge T(f_3) \wedge \\ &\quad \texttt{HYP-OF}(f_1, f_3) \wedge \texttt{HYP-OF}(\texttt{impe}(f_1, f_2), \texttt{impe}(f_1, f_3))) \\ &\qquad \supset T(\texttt{impe}(\texttt{impe}(f_1, f_2), \texttt{impe}(f_1, f_3))))\end{aligned} \tag{7}$$

This new derived inference rule may be used in subsequent proofs in the meta-theory. In the following section we show how it can be flattened back into GETFOL's Lisp source and used directly by the system for proofs in all theories.

5 Flattening

Flattening is the process of compiling formulas derived in the particular theory of implementation back down into GETFOL's Lisp source. It is the dual of lifting and the flattening functions are essentially inverses of their lifting counterparts.[3]

```
<T-RULE>   ::=  ∀ <VAR> ...  ∀ <VAR> (<T-PROP> ⊃ <T-PROP>)
<T-PROP>   ::=  (<T-PROP> ∧ <T-PROP>) |
                <PRED-CONSTANT>(<T-TERM>, ..., <T-TERM>)
<T-TERM>   ::=  <VAR> | <FUNC-CONSTANT>(<T-TERM>, ..., <T-TERM>)
```

Figure 6: Grammar for Formula Subset

Not all formulas have procedural equivalents. The grammar describing the subset of formulas that we flatten is given in Figure 6. This grammar is the dual of the Lisp subset grammar: there is an isomorphism between the syntactic categories C-RULE, C-PROP, and C-TERM, with their theory equivalents T-RULE, T-PROP, and T-TERM. Lifting, like flattening, is straightforward; the only complication is that flattening must reestablish the distinction between `IS-FACT` and `proof-add-theorem` at the source level.

Since the flattening functions given in Figure 7 are dual to the lifting functions, then if we were to flatten axioms such as `impe-rule` and `alli-rule` given in Figure 5, we would end up where we started: with their implementation given in Figure 3. By flattening derived rules, we extend the system with new procedures. Flattening Equation 7 and naming the result `impe-3` augments the system with the inference procedure given in Figure 8. Furthermore, if *name* refers to an already defined system function, then flattening $\mathcal{F}_{rule}(\forall V_1 ... \forall V_n(h \supset c)(name)$, modifies the system, replacing *name*, with the new flattened function. This provides a means for self-modification in addition to self-expansion.

Of course it is important to know that any such modifications to the prover are correct. That is, we must know that whenever $\forall V_1 ... \forall V_n(p_1 \supset p_2)$ is a theorem then extending GETFOL with $\mathcal{F}_{rule}(\forall V_1 ... \forall V_n(p_1 \supset p_2))$ will not allow the theorem prover to derive any new theorems in any theory it is applied to. This correctness requirement is stated formally and proved in [GT90c]. The proof proceeds by induction on the complexity

[3] Not quite as the formula flattening function must take an additional argument that provides a name for the function corresponding to the flattened formula.

$$
\begin{array}{l}
\mathcal{F}_{rule}(\forall V_1 ... \forall V_n (p_1 \supset p_2)(name) = \\
\quad (\texttt{DEFLAM}\ name\ (V_1 ... V_n)\ (\texttt{IF}\ \mathcal{F}_{prop_l}(p_1)\ \mathcal{F}_{prop_r}(p_2))
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{F}_{prop_l}(p_1 \wedge p_2) & = & (\texttt{AND}\ \mathcal{F}_{prop_l}(p_1)\ \mathcal{F}_{prop_l}(p_2)) \\
\mathcal{F}_{prop_l}(\texttt{T}(t)) & = & (\texttt{IS-FACT}\ \mathcal{F}_{term}(t)) \\
\mathcal{F}_{prop_l}(P(t_1, ..., t_n)) & = & (P\ \mathcal{F}_{term}(t_1)\ ...\ \mathcal{F}_{term}(t_n)) \\
 & = & \textit{for}\ P \neq T \\
\\
\mathcal{F}_{prop_r}(p_1 \wedge p_2) & = & (\texttt{AND}\ \mathcal{F}_{prop_r}(p_1)\ \mathcal{F}_{prop_r}(p_2)) \\
\mathcal{F}_{prop_r}(\texttt{T}(t)) & = & (\texttt{proof-add-theorem}\ \mathcal{F}_{term}(t)) \\
\mathcal{F}_{prop_r}(P(t_1, ..., t_n)) & = & (P\ \mathcal{F}_{term}(t_1)\ ...\ \mathcal{F}_{term}(t_n)) \\
 & = & \textit{for}\ P \neq T \\
\\
\mathcal{F}_{term}(v) & = & v \\
\mathcal{F}_{term}(f(t_1\ , ...,\ t_n)) & = & (f\ \mathcal{F}_{term}(t_1)\ ...\ \mathcal{F}_{term}(t_n))
\end{array}
$$

Figure 7: Flattening Functions

```
(DEFLAM impe-3 (f1 f2 f3)
 (IF (AND (AND (IS-FACT f1) (AND (IS-FACT f2)
     (AND (HYP-OF f1 f2) (AND (IS-FACT f1) (AND (IS-FACT f3)
     (AND (HYP-OF f1 f3) (AND (HYP-OF (impe f1 f2) (impe f1 f3))))))))))
          (proof-add-theorem (impe (impe f1 f2) (impe f1 f3)))))
```

Figure 8: Flattening of Derived Rule

of the theorem proved in the meta-theory; the various proof cases demonstrate that generated preconditions map down to tests which guarantee that the derived inference rule only proves theorems that would be provable directly using primitive inference rules.

6 Optimization

There may exist many logically equivalent formulas that flatten to procedurally distinct derived inference rules. The procedural distinction between two rules being not that they implement different inference functions (*i.e.*, generating different conclusions from identical premises), but rather their execution efficiency. The correctness of the optimized rules is a trivial consequence of the correctness of the original ones.

The simplest and most direct example of this is logical simplification. Section 4 provides such an example; there, several inference rules were combined in `GETFOL` to form a new rule. However, the combination generated redundant premises. Because both Equations 3 and 4 have $T(a_1)$ as a premise, in Equation 6, $T(a_1)$ occurs twice. However as conjunction ($\wedge$) is an associative and commutative propositional connective, and for all propositions A, $(A \wedge A) \supset A$, we can eliminate the second occurrence of $T(a_1)$ in this equation. Such a simplified formula would flatten into a function identical to the one given in

Figure 8, except there would be only one call to `(IS-FACT f1)` instead of two. Hence each application of this rule would execute faster. Furthermore, by partially axiomatizing syntactic functions, such as `HYP-OF` or `IS-FREE`, more elaborate simplification is possible by reasoning about the relationships between hypotheses.

A more complex and interesting kind of optimization is possible based on transforming the proof-structure component of derived inference rules to a direct syntactic representation of the inference function computed. Specifically, given the structure of primitive inference rules and the way that inference rules are combined into new rules, each rule amounts to a series of tests or preconditions followed by a term that constructs a new proof node (fact) as a function of some encoded proof. For example, the inference function given in Figure 8 directly encodes the proof given in Equation 2. While this construction enables us to reason about the structure of proofs at the meta-level, it can result in slowly executing code as the inference rule must mimic the proof structure in computing the new proof-node. The alternative is to replace the encoded proof-function with a direct syntactic representation of the resulting proof-node. This syntactic representation is similar to those used in systems such as Isabelle. As a simple example, consider the `impe-3` rule given in Figure 8 (flattened from the formula given in (7)). Its syntactic representation is

$$\begin{array}{l}\forall f_1 \forall f_2 \forall f_3 (T(f_1) \wedge T(f_2) \wedge T(f_3) \wedge \\ \quad \mathtt{HYP\text{-}OF}(f_1, f_2) \wedge \mathtt{HYP\text{-}OF}(f_1, f_3) \wedge \\ \quad \mathtt{concl}(\mathtt{wff\text{-}of}(f_2)) = \mathtt{hyp}(\mathtt{concl}(\mathtt{wff\text{-}of}(f_3))) \\ \qquad \supset T(\mathtt{mkfact}(\mathtt{concl}(\mathtt{concl}(\mathtt{wff\text{-}of}(f_3))), \mathtt{dep3}(f_1, f_2, f_3))))\end{array} \tag{8}$$

where $\mathtt{mkfact}(\mathit{wff}, \mathit{dep})$ builds a proof-node, *i.e.*, a formula with the assumptions it depends on, `concl` and `hyp` take the conclusion and hypothesis of their formula arguments, $\mathtt{wff\text{-}of}(\mathit{fact})$ takes the formula of a fact, and $\mathtt{dep3}(f_1, f_2, f_3)$ builds the union of the assumptions that f_1 f_2 and f_3 depend on.

Complex derived rules can be represented in this way and the representation is independent of how they were proved. That is, all the distinct deductions with the same root and leaves correspond to a unique syntactic representation. Generally, the longer the deduction based representation, the more effective the optimization is. Many more factors not described here contribute to making this operation more or less effective; some of them are: the sequence of rule applications, how introduction and elimination rules intermix, the presence of redundant steps (maximal segments [Pra65]) and so on.

These optimized syntactic inference rules can be derived in the metatheory using the appropriate axioms [GT90b]. *E.g.*,

$$\begin{array}{l}\forall f_1 \forall f_2 (\mathtt{wff\text{-}of}(\mathtt{impe}(f_1, f_2)) = \mathtt{concl}(\mathtt{wff\text{-}of}(f_2))) \\ \forall f_1 \forall f_2 (\mathtt{dep\text{-}of}(\mathtt{impe}(f_1, f_2)) = \mathtt{dep2}(f_1, f_2))\end{array} \tag{9}$$

and others. Here $\mathtt{dep\text{-}of}(\mathit{fact})$ takes the assumptions the fact depends on and $\mathtt{dep2}(f_1, f_2)$ builds the union of the assumptions that f_1 and f_2 depend on.

A very effective way to obtain these optimized representations is by partial evaluation using the `GETFOL` semantic simplifier [GT90a], which is similar to the old `FOL`'s simplifier [Wey80]. Unfortunately a further discussion of optimization via partial evaluation is beyond the scope of this paper.

7 Related Work

In GETFOL, reasoning about a logic's inference rules provides a means for deriving new rules. This idea also appears, for example, in Isabelle [Pau89], and systems with reflection principles [CAB$^+$86]. In [GT90b, GT90c], Giunchiglia and Traverso show how in GETFOL one can build a uniform hierarchy of theories where theorem proving at one level constructs new inference rules which can be used via a reflection rule at a lower level. In this report, we demonstrated that, by inverting the lifting process, we can safely flatten new derived rules back down into the underlying system. The result is that reflection principles are not required to use these new rules. Hence, our approach is very general.

We have already indicated how our work differs from those working with logic programming meta-interpreters. That is, these system have fixed control strategies that are difficult to reason about and impossible to modify; the underlying Prolog system must be treated as a black box. More closely is the work of Smith [Smi83] who defines a tower of meta-circular Lisp interpreters where modifying the description of a "Lisp processor" at one level can alter the execution of Lisp functions at lower levels. The linkage between the levels is provided by a principle of *procedural reflection*. In contrast, through our lifting and flattening functions, we do not require any kind of reflection principles nor a hierarchy of theories. Moreover, our meta-language is declarative and the "programs" which control proof-search are derived via logical deduction in a formal theory.

Within the theorem proving community, perhaps the work of Boyer and Moore on meta-functions [BM81] most closely resembles our work. In the setting of their inductive theorem prover, users can augment the system by coding term-rewriting functions in Lisp. After proving that these functions essentially preserve the "meaning" of terms, the functions become integrated into the prover and used for subsequent term simplification. Unlike our work, their system extension are effected by explicitly coded procedures as opposed to declarative reasoning about the representation of the underlying system. Their work also takes place within a rather limited logic (quantifier free first-order logic) although they may use induction principles to reason about defined functions.

8 Conclusions and Extensions

We have demonstrated how within the GETFOL theorem prover we can automatically generate a meta-theory for parts of the implementation and use this theory to safely extend GETFOL with derived meta-theoretic rules. This approach is rather general and does not involve reflection or other features of GETFOL.

Our work serves as the starting point for a deeper examination of theories of implementation. As we only lift inference rules, the kind of derived rules we can formalize and prove is limited. For example, the theorem suggested in Weyhrauch [Wey80] — a formula made up of sentential variables and the if-and-only-if connective is valid if and only if each sentential variable occurs an even number of times — would require an additional theory of syntax with appropriate induction principles.

We have recently begun to examine extensions that allow theorems like Weyhrauch's to be proven. One possibility is to use type information given in the implementation to axiomatize induction principles. For example, a type of recursively defined formulas naturally gives rise to a structural induction principle for formulas. This possibility is sketched in [BGT91] and is the subject of a forthcoming research report. We believe that such additions will enable us to extend our theories of implementation to reason about a very large class of recursive functions that should include powerful term simplifiers and decision procedures.

References

[BGT91] David Basin, Fausto Giunchiglia and Paolo Traverso. *Meta-Theory as the Dual of System Implementation.* Technical report, IRST 1991.

[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.

[GT90a] F. Giunchiglia and P. Traverso. GETFOL Manual - version 1 release 1. Technical report, IRST, 1990.

[GT90b] F. Giunchiglia and P. Traverso. Plan formation and execution in an 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.

[GT90c] F. Giunchiglia and P. Traverso. Reflective reasoning with and between a declarative metatheory and the implementation code. Technical report, IRST, 1990. IRST-Technical Report no. 9012-03 Submitted to IJCAI 91.

[Low89] T. Lownie. Extending Reflective Architectures. In *Proc. IJCAI 89*, pages 446–451, 1989.

[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.* Almquist and Wiksell, 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. Special Issue on Non-monotonic Logic*, 13(1), 1980.